# Table of Contents

# 介绍

## problem-solving-with-algorithms-and-data-structure-using-python 中文版

### 目的

数据结构作为计算机从业人员的必备基础，Java, c 之类的语言有很多这方面的书籍，Python 相对较少， 其中比较著名的一本 problem-solving-with-algorithms-and-data-structure-using-python，所以我在学习的过程中将其翻译了中文版，希望对大家有点帮助。

- 由于本人英语能力不佳，本书部分翻译参考谷歌，但每句话都经过个人理解后调整修改，尽量保证语句畅通。
- 由于翻译比较仓促，难以避免有些排版错别字等问题，后续会润色。如你也有兴趣参与，可 pull request 到 github 仓库
- 默认大家有一定的 Python 基础，故暂未翻译 Python 语法的几个章节。后续考虑书的完整性会加上这几节。
- 本书未加上课后练习，如有兴趣，可上原书网站练习。

### 地址

- github 地址: https://github.com/facert/python-data-structure-cn
- gitbook 在线浏览: https://facert.gitbooks.io/python-data-structure-cn

### 联系作者


- 邮箱： zhangcr1992@163.com
- 博客： https://facert.github.io


### 许可证

# 1.1. 目标

- 回顾计算机科学的思想, 提高编程和解决问题的能力。
- 理解抽象化以及它在解决问题过程中发挥的作用
- 理解和实现抽象数据类型的概念
- 回顾 Python 编程语言

# 1.2.快速开始

从第一台通过接入网线和交换机来传递人的指令的计算机开始，我们编程思考的方式发生了许多变化。与社会的许多方面一样，计算技术的变化为计算机科学家提供了越来越多的工具和平台来实践他们的工艺。计算机的快速发展诸如更快的处理器，高速网络和大的存储器容量已经让计算机科学家陷入高度复杂螺旋中。在所有这些快速演变中，一些基本原则保持不变。计算机科学关注用计算机来解决问题。

毫无疑问你花了相当多的时间学习解决问题的基础知识，以此希望有足够的能力把问题弄清楚并想出解决方案。你还发现编写代码通常很困难。问题的复杂性和解决方案的相应复杂性往往会掩盖与解决问题过程相关的基本思想。

本章着重介绍了其他两个重要的部分。首先回顾了计算机科学与算法和研究数据结构所必须适应的框架，特别是我们需要研究这些主题的原因，以及如何理解这些主题有助于我们更好的解决问题。第二，我们回顾 Python 编程语言。虽然我们不提供详尽的参考，我们将在其余章节中给出基本数据结构的示例和解释。

# 1.3.什么是计算机科学

计算机科学往往难以定义。这可能是由于在名称中不幸使用了“计算机”一词。正如你可能知道的，计算机科学不仅仅是计算机的研究。虽然计算机作为一个工具在学科中发挥重要的支持作用，但它们只是工具。

计算机科学是对问题，解决问题以及解决问题过程中产生的解决方案的研究。给定一个问题，计算机科学家的目标是开发一个算法，一系列的指令列表，用于解决可能出现的问题的任何实例。算法遵循它有限的过程就可以解决问题。

计算机科学可以被认为是对算法的研究。但是，我们必须谨慎地包括一些事实，即一些问题可能没有解决方案。虽然证明这种说法正确性超出了本文的范围，但一些问题不能解决的事实对于那些研究计算机科学的人是很重要的。所以我们可以这么定义计算机科学，是研究能被解决的问题的方案和不能被解决问题的科学。

通常我们会说这个问题是可计算的，当在描述问题和解决方案时。如果存在一个算法解决这个问题，那么问题是可计算的。计算机科学的另一个定义是说，计算机科学是研究那些可计算和不可计算的问题，研究是不是存在一种算法来解决它。你会注意到，“电脑”一词根本没有出现。解决方案是独立于机器而言的。

计算机科学，因为它涉及问题解决过程本身，也是抽象的研究。抽象使我们能够以分离所谓的逻辑和物理角度的方式来观察问题和解决方案。基本思想跟我们常见的例子一样。

假设你可能已经开车上学或上班。作为司机，汽车的用户。你为了让汽车载你到目的地，你会和汽车有些互动。进入汽车，插入钥匙，点火，换挡，制动，加速和转向。从抽象的角度，我们可以说你所看到的是汽车的逻辑视角。你正在使用汽车设计师提供的功能，将你从一个地方运输到另一个位置。这些功能有时也被称为接口。

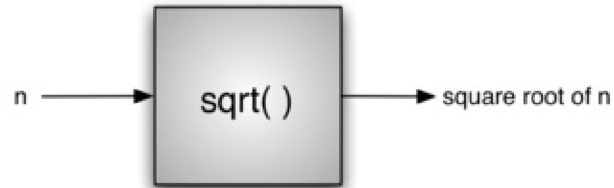
另一方面，修理汽车的技工有一个截然不同的视角。他不仅知道如何开车，还必须知道所有必要的细节，使我们认为理所当然的功能运行起来。他需要了解发动机是如何工作的，变速箱如何变速，温度是如何控制的等等。这被称为物理视角，细节发生在“引擎盖下”。

当我们使用电脑时也会发生同样的情况。大多数人使用计算机写文档，发送和接收电子邮件，上网冲浪，播放音乐，存储图像和玩游戏，而不知道让这些应用程序工作的细节。他们从逻辑或用户角度看计算机。计算机科学家，程序员，技术支持人员和系统管理员看计算机的角度截然不同。他们必须知道操作系统如何工作的细节，如何配置网络协议，以及如何编写控制功能的各种脚本。他们必须能够控制底层的细节。

这两个示例的共同点是用户态的抽象，有时也称为客户端，不需要知道细节，只要用户知道接口的工作方式。这个接口是用户与底层沟通的方式。作为抽象的另一个例子，Python 数学模块。一旦我们导入模块，我们可以执行计算

```
>>> import math
>>> math.sqrt(16)
4.0
>>>
```

这是一个程序抽象的例子。我们不一定知道如何计算平方根，但我们知道函数是什么以及如何使用它。如果我们正确地执行导入，我们可以假设该函数将为我们提供正确的结果。我们知道有人实现了平方根问题的解决方案，但我们只需要知道如何使用它。这有时被称为“黑盒子”视图。我们简单地描述下接口：函数的名称，需要什么（参数），以及将返回什么。细节隐藏在里面（见图1）。




（图1）

# 1.4.什么是编程

编程是将算法编码为符号，编程语言的过程，以使得其可以由计算机执行。虽然有许多编程语言和不同类型的计算机存在，第一步是需要有解决方案。没有算法就没有程序。

计算机科学不是研究编程。然而，编程是计算机科学家的一个重要能力。编程通常是我们为解决方案创建的表现形式。因此，这种语言表现形式和创造它的过程成为该学科的基本部分。

算法描述了依据问题实例数据所产生的解决方案和产生预期结果所需的一套步骤。编程语言必须提供一种表示方法来表示过程和数据。为此，它提供了控制结构和数据类型。

控制结构允许以方便而明确的方式表示算法步骤。至少，算法需要执行顺序处理，决策选择和重复控制迭代。只要语言提供这些基本语句，它就可以用于算法表示。

计算机中的所有数据项都以二进制形式表示。为了赋给这些字符串含义，我们需要有数据类型。数据类型提供了对这个二进制数据的解释，以便我们能够根据解决的问题思考数据。这些底层的内置数据类型（有时称为原始数据类型）为算法开发提供了基础。

例如，大多数编程语言为整数提供数据类型。内存中的二进制数据可以解释为整数，并且能给予一个我们通常与整数（例如 23,654 和 -19）相关联的含义。此外，数据类型还提供数据项参与的操作的描述。对于整数，诸如加法，减法和乘法的操作是常见的。我们期望数值类型的数据可以参与这些算术运算。通常我们遇到的困难是问题及其解决方案非常复杂。这些简单的，语言提供的结构和数据类型虽然足以表示复杂的解决方案，但通常在我们处理问题的过程中处于不利地位。我们需要一些方法控制这种复杂性，并能给我们提供更好的解决方案。

# 1.5.为什么要学习数据结构和抽象数据类型

为了管理问题的复杂性和解决问题的过程，计算机科学家使用抽象使他们能够专注于“大局”而不会迷失在细节中。通过创建问题域的模型，我们能够利用更好和更有效的问题解决过程。这些模型允许我们以更加一致的方式描述我们的算法将要处理的数据。

之前，我们将过程抽象称为隐藏特定函数的细节的过程，以允许用户或客户端在高层查看它。我们现在将注意力转向类似的思想，即数据抽象的思想。`抽象数据类型`（有时缩写为 ADT）是对我们如何查看数据和允许的操作的逻辑描述，而不用考虑如何实现它们。这意味着我们只关心数据表示什么，而不关心它最终将如何构造。通过提供这种级别的抽象，我们围绕数据创建一个封装。通过封装实现细节，我们将它们从用户的视图中隐藏。这称为信息隐藏。

Figure 2 展示了抽象数据类型是什么以及如何操作。用户与接口交互，使用抽象数据类型指定的操作。抽象数据类型是用户与之交互的 shell。实现隐藏在更深的底层。用户不关心实现的细节。

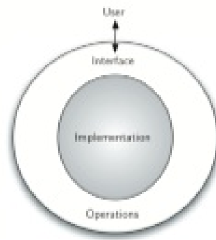



*Figure 2*

抽象数据类型（通常称为数据结构）的实现将要求我们使用一些程序构建和原始数据类型的集合来提供数据的物理视图。 正如我们前面讨论的，这两个视角的分离将允许我们将问题定义复杂的数据模型，而不给出关于模型如何实际构建的细节。 这提供了独立于实现的数据视图。由于通常有许多不同的方法来实现抽象数据类型，所以这种实现独立性允许程序员在不改变数据的用户与其交互的方式的情况下切换实现的细节。 用户可以继续专注于解决问题的过程。

# 1.6.为什么要学习算法

计算机科学家经常通过经验学习。我们通过看别人解决问题和自己解决问题来学习。接触不同的问题解决技术，看不同的算法设计有助于我们承担下一个具有挑战性的问题。通过思考许多不同的算法，我们可以开始开发模式识别，以便下一次出现类似的问题时，我们能够更好地解决它。

算法通常彼此完全不同。考虑前面看到的 `sqrt` 的例子。完全可能的是，存在许多不同的方式来实现细节以计算平方根函数。一种算法可以使用比另一种更少的资源。一个算法可能需要 10 倍的时间来返回结果。我们想要一些方法来比较这两个解决方案。即使他们都工作，一个可能比另一个“更好”。我们建议使用一个更高效，或者一个只是工作更快或使用更少的内存的算法。当我们研究算法时，我们可以学习分析技术，允许我们仅仅根据自己的特征而不是用于实现它们的程序或计算机的特征来比较和对比解决方案。

在最坏的情况下，我们可能有一个难以处理的问题，这意味着没有算法可以在实际的时间量内解决问题。重要的是能够区分具有解决方案的那些问题，不具有解决方案的那些问题，以及存在解决方案但需要太多时间或其他资源来合理工作的那些问题。

经常需要权衡，我们需要做决定。作为计算机科学家，除了我们解决问题的能力，我们还需要了解解决方案评估技术。最后，通常有很多方法来解决问题。找到一个解决方案，我们将一遍又一遍比较，然后决定它是否是一个好的方案。

# 1.7.回顾Python基础

在本节中，我们将回顾 Python 编程语言，并提供一些更详细的例子。 如果你是 Python 新手，或者你需要有关提出的任何主题的更多信息，我们建议你参考 Python语言参考或 Python 教程。我们在这里的目标是重新认识下 python 语言，并强化一些将成为后面章节中心的概念。

Python 是一种现代的，易于学习的面向对象的编程语言。 它具有一组强大的内置数据类型和易于使用的控件结构。 由于Python是 一种解释型语言，因此通过简单地查看和描述交互式会话，更容易进行检查。 你应该记得，解释器显示熟悉的 `>>>` 提示，然后计算你提供的 Python 语句。 例如，

```
>>> print("Algorithms and Data Structures")
Algorithms and Data Structures
>>>
```

显示提示，打印结果和下一个提示。

# 2.1. 目标

- 理解算法分析的重要性
- 能够使用 大$O$ 符号描述算法执行时间
- 理解 Python 列表和字典的常见操作的 大$O$ 执行时间
- 理解 Python 数据的实现是如何影响算法分析的。
- 了解如何对简单的 Python 程序做基准测试（`benchmark`）。

# 2.2.什么是算法分析

一些普遍的现象是，刚接触计算机科学的学生会将自己的程序和其他人的相比较。你可能还注意到，这些计算机程序看起来很相似，尤其是简单的程序。经常出现一个有趣的问题。当两个程序解决同样的问题，但看起来不同，哪一个更好呢？

为了回答这个问题，我们需要记住，程序和程序代表的底层算法之间有一个重要的区别。正如我们在第 1 章中所说，一种算法是一个通用的，一步一步解决某种问题的指令列表。它是用于解决一种问题的任何实例的方法，给定特定输入，产生期望的结果。另一方面，程序是使用某种编程语言编码的算法。根据程序员和他们所使用的编程语言的不同，可能存在描述相同算法的许多不同的程序。

要进一步探讨这种差异，请参考 *ActiveCode 1* 中显示的函数。这个函数解决了一个我们熟悉的问题，计算前 n 个整数的和。该算法使用初始化值为 0 的累加器（ `accumulator` ）变量。然后迭代 n 个整数，将每个依次添加到累加器。

### ActiveCode 1

```
def sumOfN(n):
    theSum = 0
    for i in range(1,n+1):
        theSum = theSum + i

    return theSum

print(sumOfN(10))
```

现在看看 *ActiveCode 2* 中的函数。乍一看，它可能很奇怪，但进一步的观察，你可以看到这个函数本质上和前一个函数在做同样的事情。不直观的原因在于编码习惯不好。我们没有使用良好的标识符（ `identifier` ）名称来提升可读性，我们在迭代步骤中使用了一个额外的赋值语句，这并不是真正必要的。

### ActiveCode 2

```
def foo(tom):
    fred = 0
    for bill in range(1,tom+1):
        barney = bill
        fred = fred + barney

    return fred

print(foo(10))
```

先前我们提出一个问题是哪个函数更好，答案取决于你的标准。如果你关注可读性，函数 *sumOfN* 肯定比 *foo* 好。事实上，你可能已经在介绍编程的课程中看到过很多例子，他们的目标之一就是帮助你编写易于阅读和理解的程序。然而，在本课程中，我们对算法本身的表示更感兴趣（当然我们希望你继续努力编写可读的，易于理解的代码）。

算法分析是基于每种算法使用的计算资源量来比较算法。我们比较两个算法，说一个比另一个算法好的原因在于它在使用资源方面更有效率，或者仅仅使用的资源更少。从这个角度来看，上面两个函数看起来很相似。它们都使用基本相同的算法来解决求和问题。

在这点上，重要的是要更多地考虑我们真正意义上的计算资源。有两种方法，一种是考虑算法解决问题所需的空间或者内存。解决方案所需的空间通常由问题本身决定。但是，有时候有的算法会有一些特殊的空间需求，这种情况下我们需要非常仔细地解释这些变动。

作为空间需求的一种替代方法，我们可以基于算法执行所需的时间来分析和比较算法。这种测量方式有时被称为算法的“执行时间”或“运行时间”。我们可以通过基准分析（`benchmark analysis`）来测量函数 *SumOfN* 的执行时间。这意味着我们将记录程序计算出结果所需的实际时间。在 Python 中，我们可以通过记录相对于系统的开始时间和结束时间来对函数进行基准测试。在 *time* 模块中有一个 *time* 函数，它可以在任意被调用的地方返回系统时钟的当前时间（以秒为单位）。通过在开始和结束的时候分别调用两次 *time* 函数，然后计算差异，就可以得到一个函数执行花费的精确秒数（大多数情况下是这样）。

### Listing 1

```
import time

def sumOfN2(n):
    start = time.time()

    theSum = 0
    for i in range(1,n+1):
        theSum = theSum + i

    end = time.time()

    return theSum, end-start
```

Listing 1 嵌入了时间函数，函数返回一个包含了执行结果和执行消耗时间的元组（`tuple`）。如果我们执行这个函数 5 次，每次计算前 10,000 个整数的和，将得到如下结果：

```
>>>for i in range(5):
       print("Sum is %d required %10.7f seconds"%sumOfN(10000))
Sum is 50005000 required  0.0018950 seconds
Sum is 50005000 required  0.0018620 seconds
Sum is 50005000 required  0.0019171 seconds
Sum is 50005000 required  0.0019162 seconds
Sum is 50005000 required  0.0019360 seconds
```

我们发现时间是相当一致的，执行这段代码平均需要0.0019秒。如果我们运行计算前 100,000 个整数的和的函数呢？

```
>>>for i in range(5):
       print("Sum is %d required %10.7f seconds"%sumOfN(100000))
Sum is 5000050000 required  0.0199420 seconds
Sum is 5000050000 required  0.0180972 seconds
Sum is 5000050000 required  0.0194821 seconds
Sum is 5000050000 required  0.0178988 seconds
Sum is 5000050000 required  0.0188949 seconds
>>>
```

再次的，尽管时间更长，但每次运行所需的时间也是非常一致的，平均大约多10倍。 对于 n 等于 1,000,000，我们得到：

```
>>>for i in range(5):
       print("Sum is %d required %10.7f seconds"%sumOfN(1000000))
Sum is 500000500000 required  0.1948988 seconds
Sum is 500000500000 required  0.1850290 seconds
Sum is 500000500000 required  0.1809771 seconds
Sum is 500000500000 required  0.1729250 seconds
Sum is 500000500000 required  0.1646299 seconds
>>>
```

在这种情况下，平均值也大约是前一次的10倍。现在考虑 *ActiveCode 3*，它显示了求解求和问题的不同方法。函数 *sumOfN3* 利用封闭方程而不是迭代来计算前n个整数的和。

$$\sum_{i=1}^{n} i = \frac{(n)(n+1)}{2}$$

### ActiveCode 3

```
def sumOfN3(n):
   return (n*(n+1))/2

print(sumOfN3(10))
```

如果我们对 *sumOfN3* 做同样的基准测试，使用 5 个不同的 n `(10,000, 100,000, 1,000,000, 10,000,000 和 100,000,000)`，我们得到如下结果

```
Sum is 50005000 required 0.00000095 seconds
Sum is 5000050000 required 0.00000191 seconds
Sum is 500000500000 required 0.00000095 seconds
Sum is 50000005000000 required 0.00000095 seconds
Sum is 5000000050000000 required 0.00000119 seconds
```

在这个输出中有两件事需要重点关注，首先上面记录的执行时间比之前任何例子都短，另外他们的执行时间和 n 无关，看起来 *sumOfN3* 几乎不受 n 的影响。

但是这个基准测试能告诉我们什么？我们可以很直观地看到使用了迭代的解决方案需要做更多的工作，因为一些程序步骤被重复执行。这可能是它需要更长时间的原因。此外，迭代方案执行所需时间随着 *n* 递增。另外还有个问题，如果我们在不同计算机上或者使用不用的编程语言运行这个函数，我们也可能得到不同的结果。如果使用老旧的计算机，可能需要更长时间才能执行完 *sumOfN3*。

我们需要一个更好的方法来描述这些算法的执行时间。基准测试计算的是程序执行的实际时间。它并不真正地提供给我们一个有用的度量（ `measurement` ），因为它取决于特定的机器，程序，时间，编译器和编程语言。 相反，我们希望有一个独立于所使用的程序或计算机的度量。这个度量将有助于独立地判断算法，并且可以用于比较不同实现方法的算法的效率。

# 2.3.大O符号

当我们试图通过执行时间来表征算法的效率时，并且独立于任何特定程序或计算机，重要的是量化算法需要的操作或者步骤的数量。选择适当的基本计算单位是个复杂的问题，并且将取决于如何实现算法。对于先前的求和算法，一个比较好的基本计算单位是对执行语句进行计数。在 sumOfN 中，赋值语句的计数为 1 `(theSum = 0)` 加上 n 的值（我们执行 `theSum=theSum+i` 的次数）。我们通过函数 T 表示 `T(n)=1 + n` 。参数 n 通常称为‘问题的规模’，我们称作 ‘T(n) 是解决问题大小为 n 所花费的时间，即 1+n 步长’。在上面的求和函数中，使用 n 来表示问题大小是有意义的。我们可以说，100,000 个整数和比 1000 个问题规模大。因此，所需时间也更长。我们的目标是表示出算法的执行时间是如何相对问题规模大小而改变的。

计算机科学家更喜欢将这种分析技术进一步扩展。事实证明，操作步骤数量不如确定 T(n) 最主要的部分来的重要。换句话说，当问题规模变大时，T(n) 函数某些部分的分量会超过其他部分。函数的数量级表示了随着 n 的值增加而增加最快的那些部分。数量级通常称为大O符号，写为 `O(f(n))` 。它表示对计算中的实际步数的近似。函数 f(n) 提供了 T(n) 最主要部分的表示方法。

在上述示例中，`T(n)=1+n` 。当 n 变大时，常数 1 对于最终结果变得越来越不重要。如果我们找的是 T(n) 的近似值，我们可以删除 1， 运行时间是 O(n)。要注意，1 对于 T(n) 肯定是重要的。但是当 n 变大时，如果没有它，我们的近似也是准确的。

另外一个示例，假设对于一些算法，确定的步数是 `T(n)=5n^2 +27n+1005` 。当 n 很小时, 例如 1 或 2 ，常数 1005 似乎是函数的主要部分。然而，随着 n 变大，n^2 这项变得越来越重要。事实上，当 n 真的很大时，其他两项在它们确定最终结果中所起的作用变得不重要。当 n 变大时，为了近似 T(n)，我们可以忽略其他项，只关注 5n^2 。系数 5 也变得不重要。我们说，T(n) 具有的数量级为 f(n)=n^2。 或者 O( n^2 ) 。

虽然我们没有在求和示例中看到这一点，但有时算法的性能取决于数据的确切值，而不是问题规模的大小。对于这种类型的算法，我们需要根据最佳情况，最坏情况或平均情况来表征它们的性能。最坏情况是指算法性能特别差的特定数据集。而相同的算法不同数据集可能具有非常好的性能。大多数情况下，算法执行效率处在两个极端之间（平均情况）。对于计算机科学家而言，重要的是了解这些区别，使它们不被某一个特定的情况误导。

当你学习算法时，一些常见的数量级函数将会反复出现。见 Table 1。为了确定这些函数中哪个是最主要的部分，我们需要看到当 n 变大的时候它们如何相互比较。

| f(n) | Name |
| --- | --- |
| \(1\) | Constant |
| \(\log n\) | Logarithmic |
| \(n\) | Linear |
| \(n\log n\) | Log Linear |
| \(n^{2}\) | Quadratic |
| \(n^{3}\) | Cubic |
| \(2^{n}\) | Exponential |

*Table 1*

Figure 1 表示了 Table 1 中的函数图。注意，当 n 很小时，函数彼此间不能很好的定义。很难判断哪个是主导的。随着 n 的增长，就有一个很明确的关系，很容易看出它们之间的大小关系。

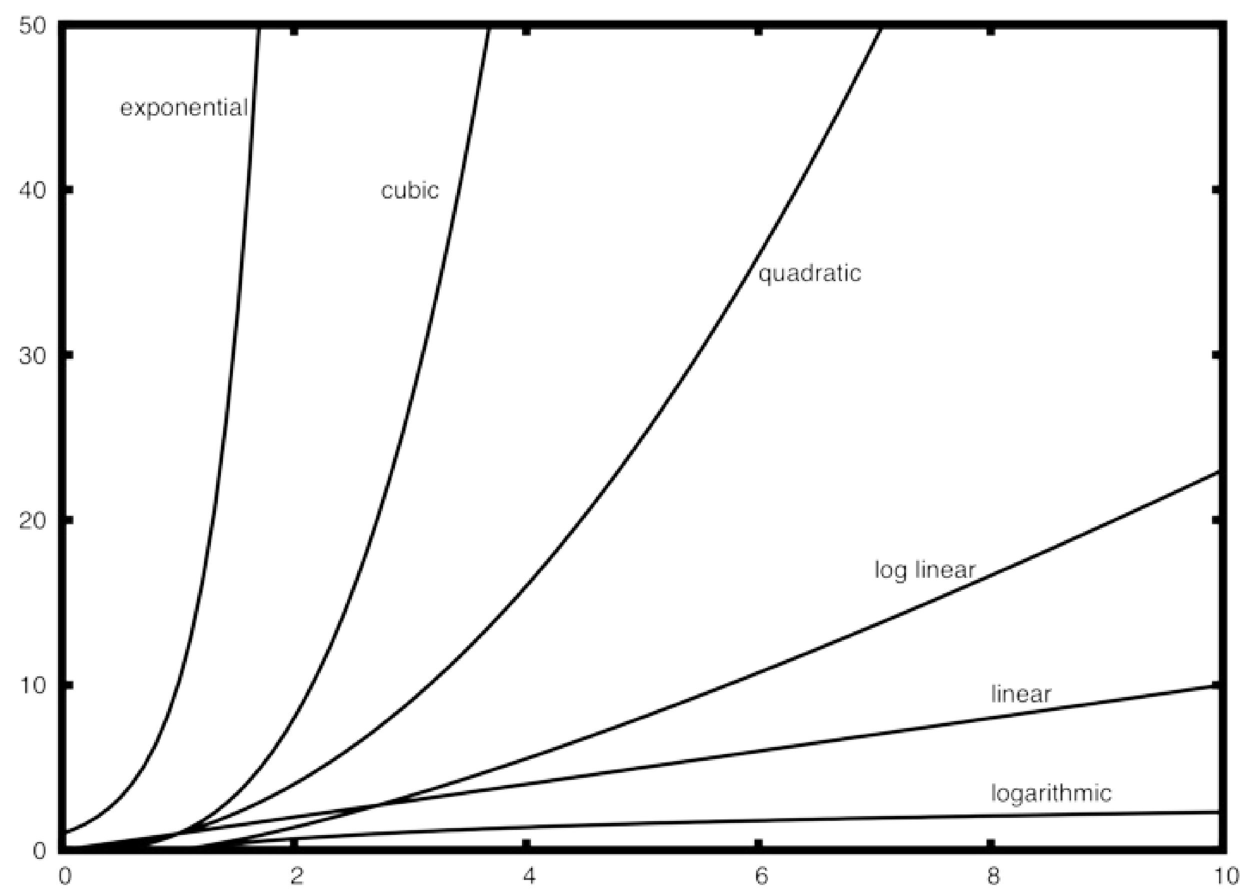



*Figure 1*

最后一个例子，假设我们有 Listing2 的代码段。虽然这个程序没有做任何事，但是对我们获取实际的代码和性能分析是有益的。

```
a=5
b=6
c=10
for i in range(n):
   for j in range(n):
      x = i * i
      y = j * j
      z = i * j
for k in range(n):
   w = a*k + 45
   v = b*b
d = 33
```

*Listing 2*

分配操作数分为四个项的总和。第一个项是常数 3, 表示片段开始的三个赋值语句。第二项是 3n^2, 因为由于嵌套迭代，有三个语句执行 n^2 次。第三项是 2n, 两个语句迭代 n 次。最后，第四项是常数 1，表示最终赋值语句。最后得出 T(n)=3+3n^2 +2n+1=3n^2 + 2n+4，通过查看指数，我们可以看到 n^2 项是显性的，因此这个代码段是 O(n^ 2 )。当 n 增大时，所有其他项以及主项上的系数都可以忽略。

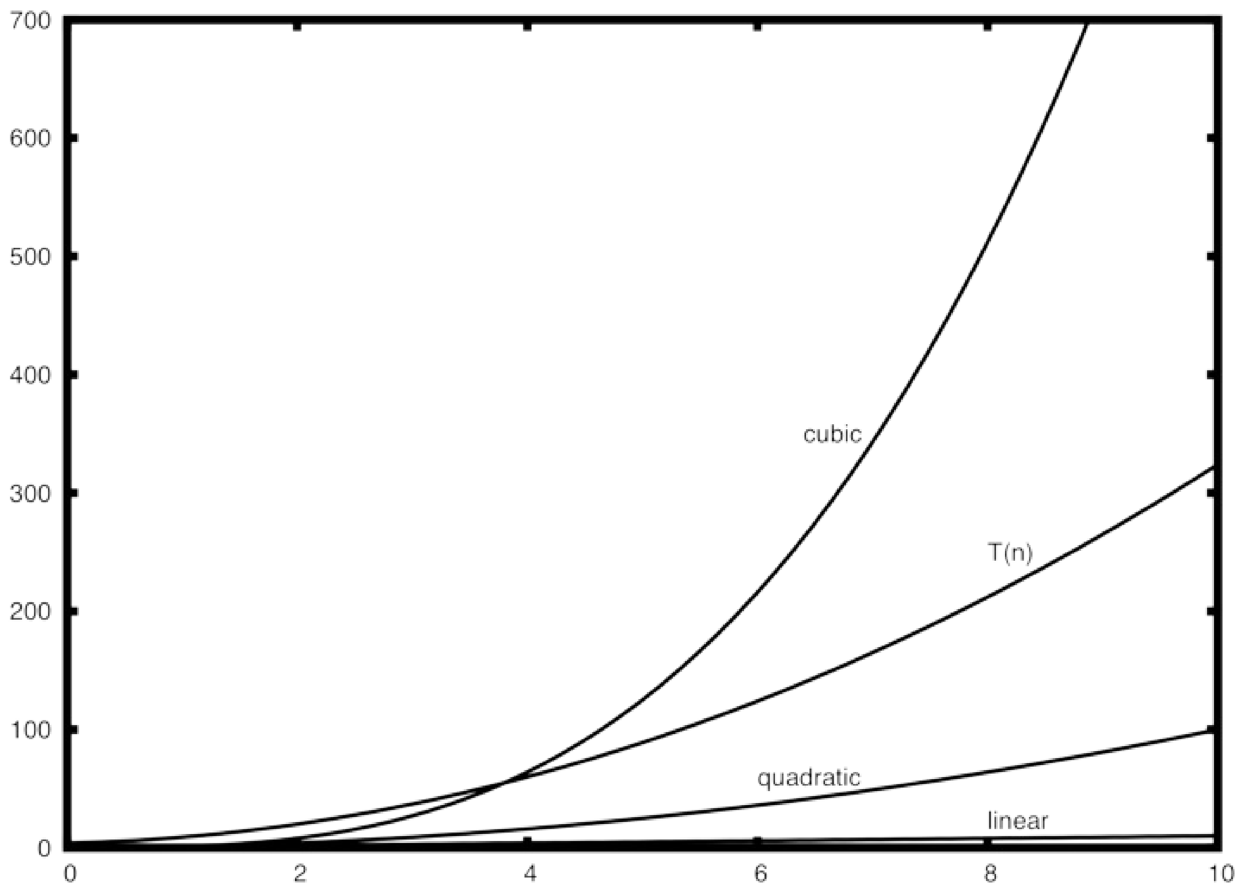



*Figure 2*

Figure 2 展示了一些常用的 大O 函数，跟上面讨论的 T(n) 函数比较，一开始的时候，T(n) 大于三次函数，后来随着 n 的增长，三次函数超过了 T(n)。T(n) 随着二次函数继续增长。

# 2.4.一个乱序字符串检查的例子

显示不同量级的算法的一个很好的例子是字符串的乱序检查。乱序字符串是指一个字符串只是另一个字符串的重新排列。例如，'heart' 和 'earth' 就是乱序字符串。'python' 和 'typhon' 也是。为了简单起见，我们假设所讨论的两个字符串具有相等的长度，并且他们由 26 个小写字母集合组成。我们的目标是写一个布尔函数，它将两个字符串做参数并返回它们是不是乱序。

## 2.4.1.解法1:检查

我们对乱序问题的第一个解法是检查第一个字符串是不是出现在第二个字符串中。如果可以检验到每一个字符，那这两个字符串一定是乱序。可以通过用 None 替换字符来了解一个字符是否完成检查。但是，由于 Python 字符串是不可变的，所以第一步是将第二个字符串转换为列表。检查第一个字符串中的每个字符是否存在于第二个列表中，如果存在，替换成 None。见 ActiveCode1

```
def anagramSolution1(s1,s2):
    alist = list(s2)

    pos1 = 0
    stillOK = True

    while pos1 < len(s1) and stillOK:
        pos2 = 0
        found = False
        while pos2 < len(alist) and not found:
            if s1[pos1] == alist[pos2]:
                found = True
            else:
                pos2 = pos2 + 1

        if found:
            alist[pos2] = None
        else:
            stillOK = False

        pos1 = pos1 + 1

    return stillOK

print(anagramSolution1('abcd','dcba'))
```

*ActiveCode1*

为了分析这个算法，我们注意到 s1 的每个字符都会在 s2 中进行最多 n 个字符的迭代。s2 列表中的 n 个位置将被访问一次来匹配来自 s1 的字符。访问次数可以写成 1 到 n 整数的和，可以写成

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$$

当 n 变大，n^2 这项占据主导，1/2 可以忽略。所以这个算法复杂度为 O(n^2 )。

## 2.4.2.解法2:排序和比较

另一个解决方案是利用这么一个事实：即使 s1,s2 不同，它们都是由完全相同的字符组成的。所以，我们按照字母顺序从 a 到 z 排列每个字符串，如果两个字符串相同，那这两个字符串就是乱序字符串。见 ActiveCode2。

```
def anagramSolution2(s1,s2):
    alist1 = list(s1)
    alist2 = list(s2)

    alist1.sort()
    alist2.sort()

    pos = 0
    matches = True

    while pos < len(s1) and matches:
        if alist1[pos]==alist2[pos]:
            pos = pos + 1
        else:
            matches = False

    return matches

print(anagramSolution2('abcde','edcba'))
```

*ActiveCode2*

首先你可能认为这个算法是 O(n)，因为只有一个简单的迭代来比较排序后的 n 个字符。但是，调用 Python 排序不是没有成本。正如我们将在后面的章节中看到的，排序通常是 O(n^2) 或 O(nlogn)。所以排序操作比迭代花费更多。最后该算法跟排序过程有同样的量级。

## 2.4.3.解法3: 穷举法

解决这类问题的强力方法是穷举所有可能性。对于乱序检测，我们可以生成 s1 的所有乱序字符串列表，然后查看是不是有 s2。这种方法有一点困难。当 s1 生成所有可能的字符串时，第一个位置有 n 种可能，第二个位置有 n-1 种，第三个位置有 n-3 种，等等。总数为 n∗(n−1)∗(n−2)∗...∗3∗2∗1n∗(n−1)∗(n−2)∗...∗3∗2∗1， 即 n!。虽然一些字符串可能是重复的，程序也不可能提前知道这样，所以他仍然会生成 n! 个字符串。

事实证明，n! 比 n^2 增长还快，事实上，如果 s1 有 20个字符长，则将有 20! = 2,432,902,008,176,640,000 个字符串产生。如果我们每秒处理一种可能字符串，那么需要 77,146,816,596 年才能过完整个列表。所以这不是很好的解决方案。

## 2.4.4.解法4: 计数和比较

我们最终的解决方法是利用两个乱序字符串具有相同数目的 a, b, c 等字符的事实。我们首先计算的是每个字母出现的次数。由于有 26 个可能的字符，我们就用 一个长度为 26 的列表，每个可能的字符占一个位置。每次看到一个特定的字符，就增加该位置的计数器。最后如果两个列表的计数器一样，则字符串为乱序字符串。见 ActiveCode 3

```
def anagramSolution4(s1,s2):
    c1 = [0]*26
    c2 = [0]*26

    for i in range(len(s1)):
        pos = ord(s1[i])-ord('a')
        c1[pos] = c1[pos] + 1

    for i in range(len(s2)):
        pos = ord(s2[i])-ord('a')
        c2[pos] = c2[pos] + 1

    j = 0
    stillOK = True
    while j<26 and stillOK:
        if c1[j]==c2[j]:
            j = j + 1
        else:
            stillOK = False

    return stillOK

print(anagramSolution4('apple','pleap'))
```

*ActiveCode 3*

同样，这个方案有多个迭代，但是和第一个解法不一样，它不是嵌套的。两个迭代都是 n, 第三个迭代，比较两个计数列表，需要 26 步，因为有 26 个字母。一共 T(n)=2n+26T(n)=2n+26，即 O(n)，我们找到了一个线性量级的算法解决这个问题。

在结束这个例子之前，我们来讨论下空间花费，虽然最后一个方案在线性时间执行，但它需要额外的存储来保存两个字符计数列表。换句话说，该算法牺牲了空间以获得时间。

很多情况下，你需要在空间和时间之间做出权衡。这种情况下，额外空间不重要，但是如果有数百万个字符，就需要关注下。作为一个计算机科学家，当给定一个特定的算法，将由你决定如何使用计算资源。

# 2.5.Python数据结构的性能

现在你对 大O 算法和不同函数之间的差异有了了解。本节的目标是告诉你 Python 列表和字典操作的 大O 性能。然后我们将做一些基于时间的实验来说明每个数据结构的花销和使用这些数据结构的好处。重要的是了解这些数据结构的效率，因为它们是本书实现其他数据结构所用到的基础模块。本节中，我们将不会说明为什么是这个性能。在后面的章节中，你将看到列表和字典一些可能的实现，以及性能是如何取决于实现的。

# 2.6.列表

python 的设计者在实现列表数据结构的时候有很多选择。每一个这种选择都可能影响列表操作的性能。为了帮助他们做出正确的选择，他们查看了最常使用列表数据结构的方式，并且优化了实现，以便使得最常见的操作非常快。当然，他们还试图使较不常见的操作快速，但是当需要做出折衷时，较不常见的操作的性能通常牺牲以支持更常见的操作。

两个常见的操作是索引和分配到索引位置。无论列表有多大，这两个操作都需要相同的时间。当这样的操作和列表的大小无关时，它们是 O（1）。

另一个非常常见的编程任务是增加一个列表。有两种方法可以创建更长的列表，可以使用 append 方法或拼接运算符。append 方法是 O（1)。 然而，拼接运算符是 O（k），其中 k 是要拼接的列表的大小。这对你来说很重要，因为它可以帮助你通过选择合适的工具来提高你自己的程序的效率。

让我们看看四种不同的方式，我们可以生成一个从0开始的n个数字的列表。首先，我们将尝试一个 for 循环并通过创建列表，然后我们将使用 append 而不是拼接。接下来，我们使用列表生成器创建列表，最后，也是最明显的方式，通过调用列表构造函数包装 range 函数。

```
def test1():
    l = []
    for i in range(1000):
        l = l + [i]

def test2():
    l = []
    for i in range(1000):
        l.append(i)

def test3():
    l = [i for i in range(1000)]

def test4():
    l = list(range(1000))
```

要捕获我们的每个函数执行所需的时间，我们将使用 Python 的 timeit 模块。timeit 模块旨在允许 Python 开发人员通过在一致的环境中运行函数并使用尽可能相似的操作系统的时序机制来进行跨平台时序测量。

要使用 timeit，你需要创建一个 Timer 对象，其参数是两个 Python 语句。第一个参数是一个你想要执行时间的 Python 语句; 第二个参数是一个将运行一次以设置测试的语句。然后 timeit 模块将计算执行语句所需的时间。默认情况下，timeit 将尝试运行语句一百万次。 当它完成

时，它返回时间作为表示总秒数的浮点值。由于它执行语句一百万次，可以读取结果作为执行测试一次的微秒数。你还可以传递 timeit 一个参数名字为 number，允许你指定执行测试语句的次数。以下显示了运行我们的每个测试功能 1000 次需要多长时间。

```
t1 = Timer("test1()", "from __main__ import test1")
print("concat ",t1.timeit(number=1000), "milliseconds")
t2 = Timer("test2()", "from __main__ import test2")
print("append ",t2.timeit(number=1000), "milliseconds")
t3 = Timer("test3()", "from __main__ import test3")
print("comprehension ",t3.timeit(number=1000), "milliseconds")
t4 = Timer("test4()", "from __main__ import test4")
print("list range ",t4.timeit(number=1000), "milliseconds")

concat  6.54352807999 milliseconds
append  0.306292057037 milliseconds
comprehension  0.147661924362 milliseconds
list range  0.0655000209808 milliseconds
```

在上面的例子中，我们对 test1(), test2() 等的函数调用计时，setup 语句可能看起来很奇怪，所以我们详细说明下。你可能非常熟悉 from ,import 语句，但这通常用在 python 程序文件的开头。在这种情况下，`from __main__ import test1` 从 `__main__` 命名空间导入到 timeit 设置的命名空间中。timeit 这么做是因为它想在一个干净的环境中做测试，而不会因为可能有你创建的任何杂变量，以一种不可预见的方式干扰你函数的性能。

从上面的试验清楚的看出，append 操作比拼接快得多。其他两种方法，列表生成器的速度是 append 的两倍。

最后一点，你上面看到的时间都是包括实际调用函数的一些开销，但我们可以假设函数调用开销在四种情况下是相同的，所以我们仍然得到的是有意义的比较。因此，拼接字符串操作需要 6.54 毫秒并不准确，而是拼接字符串这个函数需要 6.54 毫秒。你可以测试调用空函数所需要的时间，并从上面的数字中减去它。

现在我们已经看到了如何具体测试性能，见 Table2, 你可能想知道 pop 两个不同的时间。当列表末尾调用 pop 时，它需要 O(1), 但是当在列表中第一个元素或者中间任何地方调用 pop, 它是 O(n)。原因在于 Python 实现列表的方式，当一个项从列表前面取出，列表中的其他元素靠近起始位置移动一个位置。你会看到索引操作为 O(1)。python的实现者会权衡选择一个好的方案。

| Operation | Big-O Efficiency |
| --- | --- |
| index [] | O(1) |
| index assignment | O(1) |
| append | O(1) |
| pop() | O(1) |
| pop(i) | O(n) |
| insert(i,item) | O(n) |
| del operator | O(n) |
| iteration | O(n) |
| contains (in) | O(n) |
| get slice [x:y] | O(k) |
| del slice | O(n) |
| set slice | O(n+k) |
| reverse | O(n) |
| concatenate | O(k) |
| sort | O(n log n) |
| multiply | O(nk) |

作为一种演示性能差异的方法，我们用 timeit 来做一个实验。我们的目标是验证从列表从末尾 pop 元素和从开始 pop 元素的性能。同样，我们也想测量不同列表大小对这个时间的影响。我们期望看到的是，从列表末尾处弹出所需时间将保持不变，即使列表不断增长。而从列表开始处弹出元素时间将随列表增长而增加。

Listing 4 展示了两种 pop 方式的比较。从第一个示例看出，从末尾弹出需要 0.0003 毫秒。从开始弹出要花费 4.82 毫秒。对于一个 200 万的元素列表，相差 16000 倍。

Listing 4 需要注意的几点，第一， `from __main__ import x` , 虽然我们没有定义一个函数，我们确实希望能够在我们的测试中使用列表对象 x, 这种方法允许我们只计算单个弹出语句，获得该操作最精确的测量时间。因为 timer 重复了 1000 次，该列表每次循环大小都减 1。但是由于初始列表大小为 200万，我们只减少总体大小的 0.05%。

```
popzero = timeit.Timer("x.pop(0)",
                       "from __main__ import x")
popend = timeit.Timer("x.pop()",
                      "from __main__ import x")

x = list(range(2000000))
popzero.timeit(number=1000)
4.8213560581207275

x = list(range(2000000))
popend.timeit(number=1000)
0.0003161430358886719
```

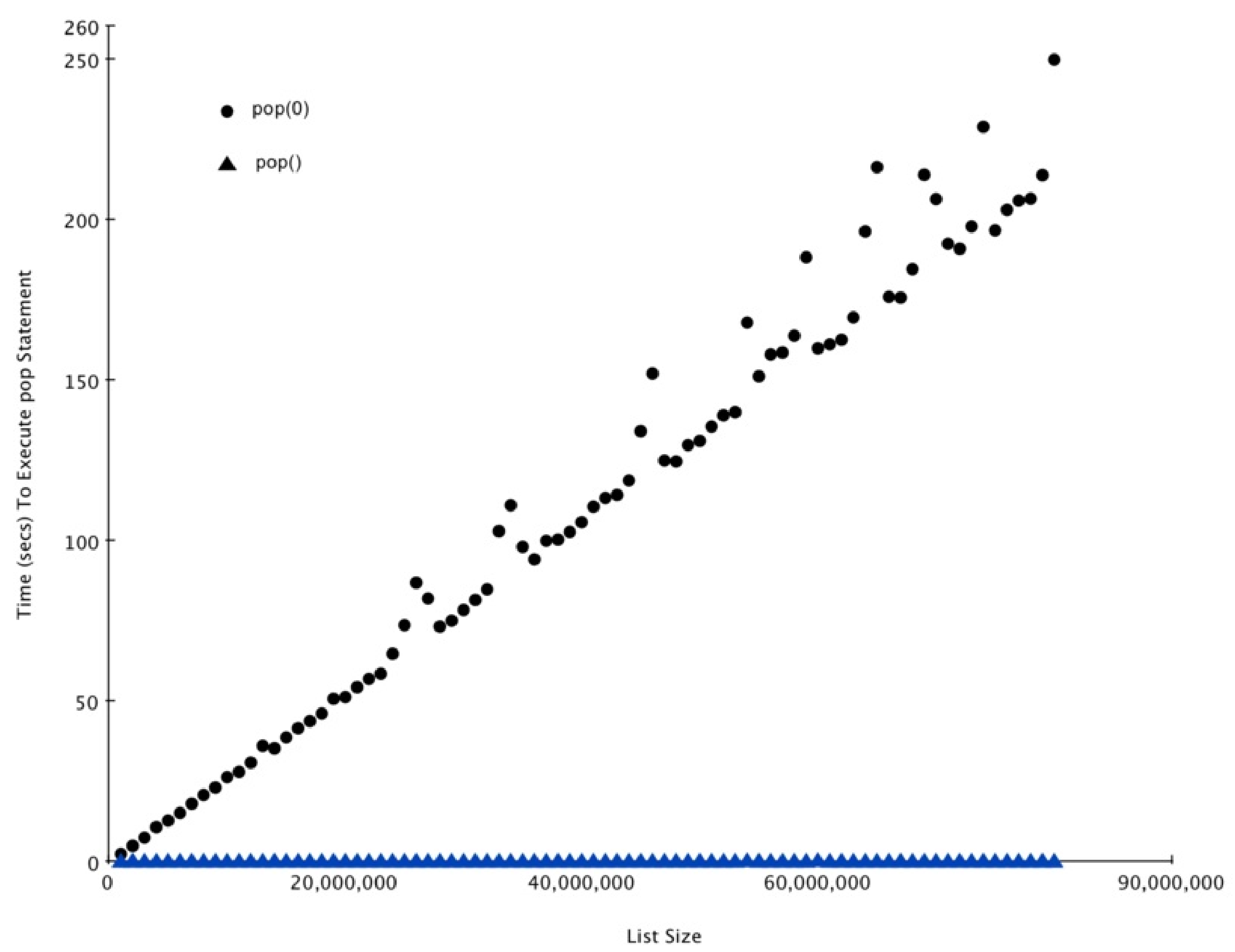
*Listing 4*

虽然我们第一个测试显示 pop(0) 比 pop() 慢， 但它没有证明 pop(0) 是 O(n), pop()是 O(1)。要验证它，我们需要看下一系列列表大小的调用效果。

```
popzero = Timer("x.pop(0)",
                "from __main__ import x")
popend = Timer("x.pop()",
               "from __main__ import x")
print("pop(0)   pop()")
for i in range(1000000,100000001,1000000):
    x = list(range(i))
    pt = popend.timeit(number=1000)
    x = list(range(i))
    pz = popzero.timeit(number=1000)
    print("%15.5f, %15.5f" %(pz,pt))
```

*Listing 5*

Figure 3 展示了我们实验的结果，你可以看到，随着列表变长，pop(0) 时间也增加，而 pop() 时间保持非常平坦。这正是我们期望看到的 O(n)和 O(1)


pop(0)
pop()
Time (secs) To Execute pop Statement
List Size
260
250
200
150
100
50
0
20,000,000
40,000,000
60,000,000
90,000,000

# 2.7.字典

python 中第二个主要的数据结构是字典。你可能记得，字典和列表不同，你可以通过键而不是位置来访问字典中的项目。在本书的后面，你会看到有很多方法来实现字典。字典的 get 和 set 操作都是 O(1)。另一个重要的操作是 contains，检查一个键是否在字典中也是 O(1)。所有字典操作的效率总结在 Table3 中。关于字典性能的一个重要方面是，我们在表中提供的效率是针对平均性能。 在一些罕见的情况下，contains，get item 和 set item 操作可以退化为 O(n)。我们将在后面的章节介绍。

| operation | Big-O Efficiency |
|---|---|
| copy | O(n) |
| get item | O(1) |
| set item | O(1) |
| delete item | O(1) |
| contains (in) | O(1) |
| iteration | O(n) |

*Table 3*

我们会在最后的实验中，将比较列表和字典之间的 contains 操作的性能。在此过程中，我们将确认列表的 contains 操作符是 O(n)，字典的 contains 操作符是 O(1)。我们将在实验中列出一系列数字。然后随机选择数字，并检查数字是否在列表中。如果我们的性能表是正确的，列表越大，确定列表中是否包含任意一个数字应该花费的时间越长。

Listing 6 实现了这个比较。注意，我们对容器中的数字执行完全相同的操作。区别在于在第 7 行上 x 是一个列表，第9行上的 x 是一个字典。

```
import timeit
import random

for i in range(10000,1000001,20000):
    t = timeit.Timer("random.randrange(%d) in x"%i,
                     "from __main__ import random,x")
    x = list(range(i))
    lst_time = t.timeit(number=1000)
    x = {j:None for j in range(i)}
    d_time = t.timeit(number=1000)
    print("%d,%10.3f,%10.3f" % (i, lst_time, d_time))
```

*Listing 6*

Figure 4 展示了 Listing6 的结果。你可以看到字典一直更快。 对于最小的列表大小为10,000个元素，字典是列表的89.4倍。对于最大的列表大小为990,000 个元素。字典是列表的11,603倍！你还可以看到列表上的contains运算符所花费的时间与列表的大小成线性增长。这验证了列表上的contains运算符是 O(n) 的断言。还可以看出，字典中的 contains 运算符的时间是恒定的，即使字典大小不断增长。事实上，对于字典大小为10,000个元素，contains操作占用0.004毫秒，对于字典大小为990,000个元素，它也占用0.004毫秒。

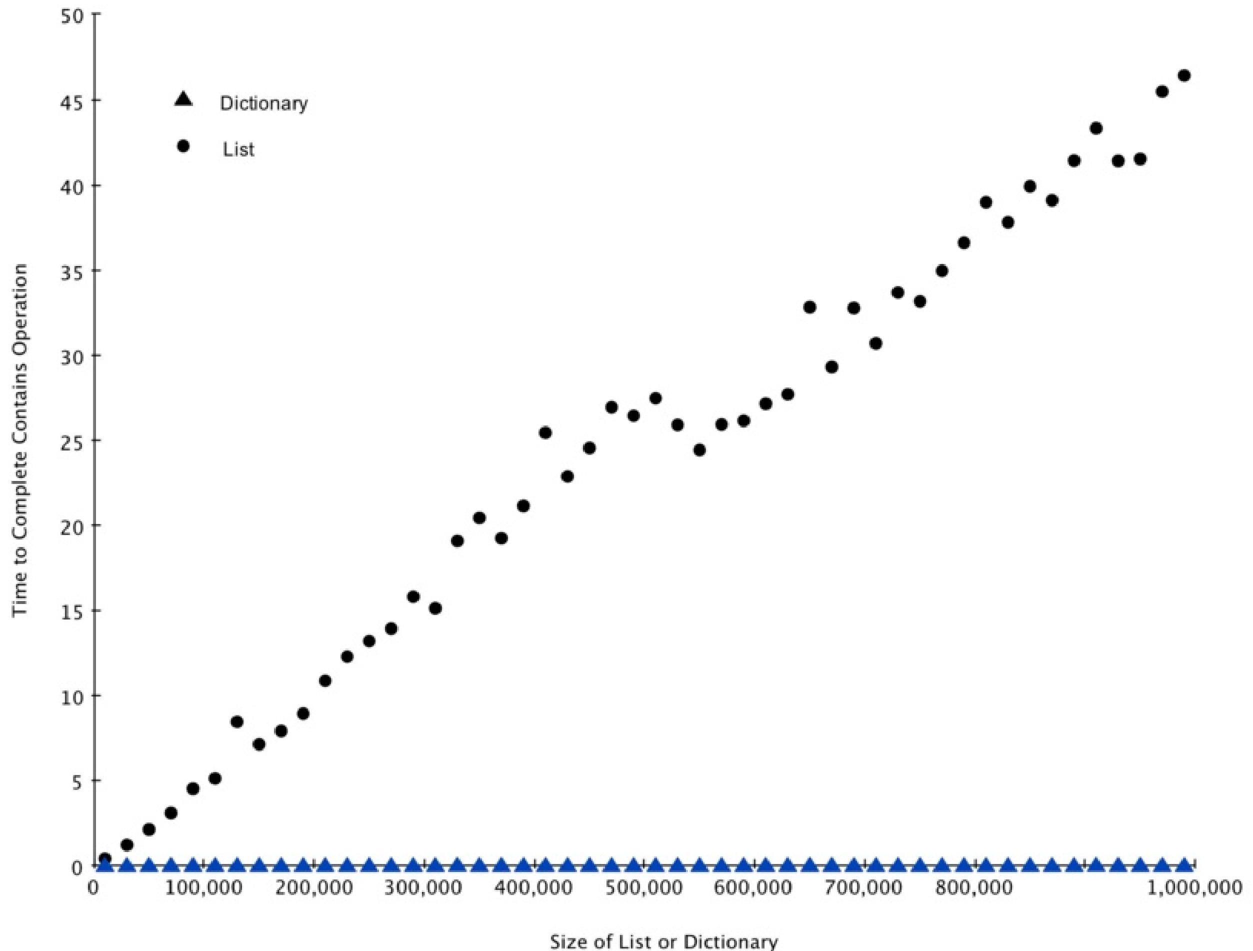



*Figure 4*

由于 Python 是一种不断发展的语言，底层总是有变化的。 有关 Python 数据结构性能的最新信息可以在 Python 网站上找到。 在撰写本文时，Python wiki有一个很好的时间复杂性页面，可以在 Time Complexity Wiki 中找到。

# 2.8.总结

- 算法分析是一种独立的测量算法的方法。
- 大O表示法允许根据问题的大小，通过其主要部分来对算法进行分类。

# 3.1. 目标

- 理解抽象数据类型的栈，队列，deque 和列表。
- 能够使用 Python 列表实现 ADT 堆栈，队列和 deque。
- 了解基本线性数据结构实现的性能。
- 了解前缀，中缀和后缀表达式格式。
- 使用栈来实现后缀表达式。
- 使用栈将表达式从中缀转换为后缀。
- 使用队列进行基本时序仿真。
- 能够识别问题中栈，队列和 deques 数据结构的适当使用。
- 能够使用节点和引用将抽象数据类型列表实现为链表。
- 能够比较我们的链表实现与 Python 的列表实现的性能。

# 3.2.什么是线性数据结构

我们从四个简单但重要的概念开始研究数据结构。栈，队列，deques, 列表是一类数据的容器，它们数据项之间的顺序由添加或删除的顺序决定。一旦一个数据项被添加，它相对于前后元素一直保持该位置不变。诸如此类的数据结构被称为线性数据结构。

线性数据结构有两端，有时被称为左右，某些情况被称为前后。你也可以称为顶部和底部，名字都不重要。将两个线性数据结构区分开的方法是添加和移除项的方式，特别是添加和移除项的位置。例如一些结构允许从一端添加项，另一些允许从另一端移除项。

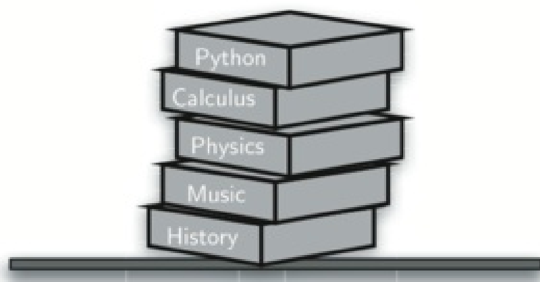
这些变种的形式产生了计算机科学最有用的数据结构。他们出现在各种算法中，并可以用于解决很多重要的问题。

# 3.3.什么是栈

栈（有时称为“后进先出栈”）是一个项的有序集合，其中添加移除新项总发生在同一端。这一端通常称为“顶部”。与顶部对应的端称为“底部”。

栈的底部很重要，因为在栈中靠近底部的项是存储时间最长的。最近添加的项是最先会被移除的。这种排序原则有时被称为 LIFO，后进先出。它基于在集合内的时间长度做排序。较新的项靠近顶部，较旧的项靠近底部。

栈的例子很常见。几乎所有的自助餐厅都有一堆托盘或盘子，你从顶部拿一个，就会有一个新的托盘给下一个客人。想象桌上有一堆书(Figure 1), 只有顶部的那本书封面可见，要看到其他书的封面，只有先移除他们上面的书。Figure 2 展示了另一个栈，包含了很多 Python 对象。



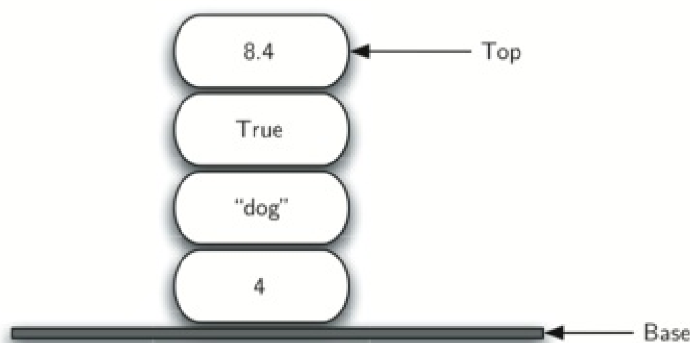




*Figure 1*

*Figure 2*

和栈相关的最有用的想法之一来自对它的观察。假设从一个干净的桌面开始，现在把书一本本叠起来，你在构造一个栈。考虑下移除一本书会发生什么。移除的顺序跟刚刚被放置的顺序相反。栈之所以重要是因为它能反转项的顺序。插入跟删除顺序相反，Figure 3 展示了 Python 数据对象创建和删除的过程，注意观察他们的顺序。

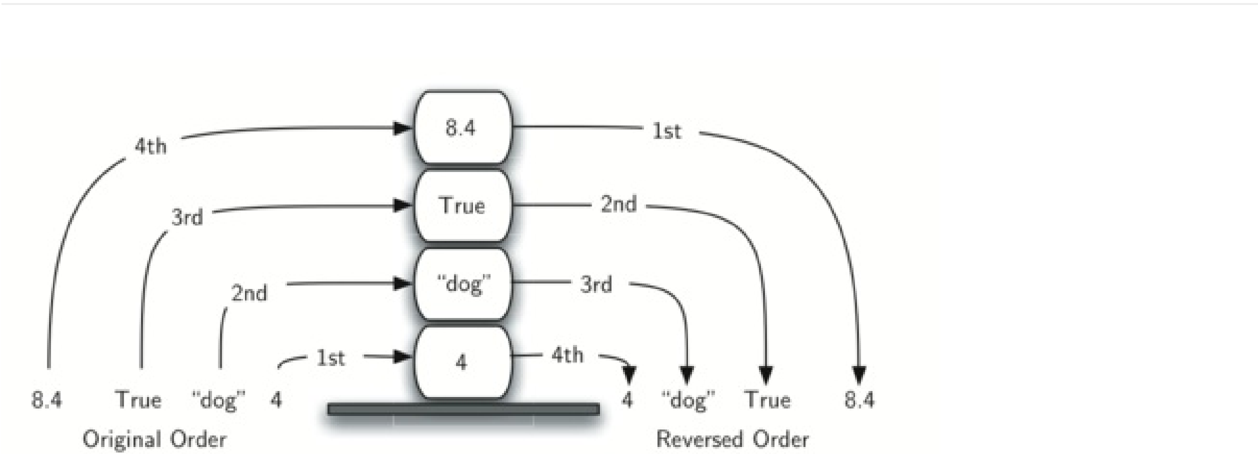



*Figure 3*

想想这种反转的属性，你可以想到使用计算机的时候所碰到的例子。例如，每个 web 浏览器都有一个返回按钮。当你浏览网页时，这些网页被放置在一个栈中（实际是网页的网址）。你现在查看的网页在顶部，你第一个查看的网页在底部。如果按'返回'按钮，将按相反的顺序浏览刚才的页面。

# 3.4.栈的抽象数据类型

栈的抽象数据类型由以下结构和操作定义。如上所述，栈被构造为项的有序集合，其中项被添加和从末端移除的位置称为“顶部”。栈是有序的 LIFO 。栈操作如下。

- Stack() 创建一个空的新栈。 它不需要参数，并返回一个空栈。
- push(item)将一个新项添加到栈的顶部。它需要 item 做参数并不返回任何内容。
- pop() 从栈中删除顶部项。它不需要参数并返回 item 。栈被修改。
- peek() 从栈返回顶部项，但不会删除它。不需要参数。 不修改栈。
- isEmpty() 测试栈是否为空。不需要参数，并返回布尔值。
- size() 返回栈中的 item 数量。不需要参数，并返回一个整数。

例如，s 是已经创建的空栈，Table1 展示了栈操作序列的结果。栈中，顶部项列在最右边。

| Stack Operation | Stack Contents | Return Value |
|---|---|---|
| `s.isEmpty()` | `[]` | `True` |
| `s.push(4)` | `[4]` | |
| `s.push('dog')` | `[4,'dog']` | |
| `s.peek()` | `[4,'dog']` | `'dog'` |
| `s.push(True)` | `[4,'dog',True]` | |
| `s.size()` | `[4,'dog',True]` | `3` |
| `s.isEmpty()` | `[4,'dog',True]` | `False` |
| `s.push(8.4)` | `[4,'dog',True,8.4]` | |
| `s.pop()` | `[4,'dog',True]` | `8.4` |
| `s.pop()` | `[4,'dog']` | `True` |
| `s.size()` | `[4,'dog']` | `2` |

*Table1*

# 3.5.Python实现栈

现在我们已经将栈清楚地定义了抽象数据类型，我们将把注意力转向使用 Python 实现栈。回想一下，当我们给抽象数据类型一个物理实现时，我们将实现称为数据结构。

正如我们在第1章中所描述的，在 Python 中，与任何面向对象编程语言一样，抽象数据类型（如栈）的选择的实现是创建一个新类。栈操作实现为类的方法。此外，为了实现作为元素集合的栈，使用由 Python 提供的原语集合的能力是有意义的。 我们将使用列表作为底层实现。

回想一下，Python 中的列表类提供了有序集合机制和一组方法。例如，如果我们有列表 [2,5,3,6,7,4]，我们只需要确定列表的哪一端将被认为是栈的顶部。一旦确定，可以使用诸如 append 和 pop 的列表方法来实现操作。

以下栈实现（ActiveCode 1）假定列表的结尾将保存栈的顶部元素。随着栈增长（push 操作），新项将被添加到列表的末尾。 pop 也操作列表末尾的元素。

```
class Stack:
     def __init__(self):
         self.items = []

     def isEmpty(self):
         return self.items == []

     def push(self, item):
         self.items.append(item)

     def pop(self):
         return self.items.pop()

     def peek(self):
         return self.items[len(self.items)-1]

     def size(self):
         return len(self.items)
```

*ActiveCode 1*

记住我们只定义类的实现，我们需要创建一个栈，然后使用它。ActiveCode 2 展示了我们通过实例化 Stack 类执行 Table 1中的操作。注意，Stack 类的定义是从 pythonds 模块导入的。

> **Note** pythonds 模块包含本书中讨论的所有数据结构的实现。它根据以下部分构造：基本数据类型，树和图。 该模块可以从 pythonworks.org下载。

```
from pythonds.basic.stack import Stack

s=Stack()

print(s.isEmpty())
s.push(4)
s.push('dog')
print(s.peek())
s.push(True)
print(s.size())
print(s.isEmpty())
s.push(8.4)
print(s.pop())
print(s.pop())
print(s.size())
```

*ActiveCode 2*

# 3.6.简单括号匹配

我们现在把注意力转向使用栈解决真正的计算机问题。你会这么写算术表达式

`(5+6)*(7+8)/(4+3)`

其中括号用于命令操作的执行。你可能也有一些语言的经验，如 Lisp 的构造

```
(defun square(n)
     (* n n))
```

这段代码定义了一个名为 square 的函数，它将返回参数的 n 的平方。 Lisp 使用大量的圆括号是臭名昭著的。

在这两个例子中，括号必须以匹配的方式出现。括号匹配意味着每个开始符号具有相应的结束符号，并且括号能被正确嵌套。考虑下面正确匹配的括号字符串：

```
(()()()())

(((())))

(()((())()))
```

对比那些不匹配的括号：

```
((((((())

()))

(()()(()
```

区分括号是否匹配的能力是识别很多编程语言结构的重要部分。具有挑战的是如何编写一个算法，能够从左到右读取一串符号，并决定符号是否平衡。为了解决这个问题，我们需要做一个重要的观察。从左到右处理符号时，最近开始符号必须与下一个关闭符号相匹配(见 Figure 4)。此外，处理的第一个开始符号必须等待直到其匹配最后一个符号。结束符号以相反的顺序匹配开始符号。他们从内到外匹配。这是一个可以用栈解决问题的线索。

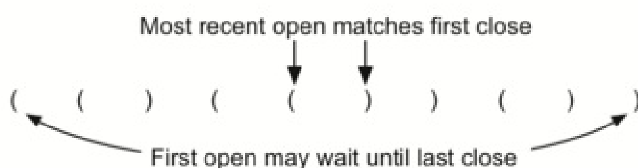


*Figure 4*

一旦你认为栈是保存括号的恰当的数据结构，算法是很直接的。从空栈开始，从左到右处理括号字符串。如果一个符号是一个开始符号，将其作为一个信号，对应的结束符号稍后会出现。另一方面，如果符号是结束符号，弹出栈，只要弹出栈的开始符号可以匹配每个结束符号，则括号保持匹配状态。如果任何时候栈上没有出现符合开始符号的结束符号，则字符串不匹配。最后，当所有符号都被处理后，栈应该是空的。实现此算法的 Python 代码见 ActiveCode 1。

```
from pythonds.basic.stack import Stack

def parChecker(symbolString):
    s = Stack()
    balanced = True
    index = 0
    while index < len(symbolString) and balanced:
        symbol = symbolString[index]
        if symbol == "(":
            s.push(symbol)
        else:
            if s.isEmpty():
                balanced = False
            else:
                s.pop()

        index = index + 1

    if balanced and s.isEmpty():
        return True
    else:
        return False

print(parChecker('((()))'))
print(parChecker('(()'))
```

*ActiveCode 1*

# 3.7.符号匹配

上面显示的匹配括号问题是许多编程语言都会出现的一般情况的特定情况。匹配和嵌套不同种类的开始和结束符号的情况经常发生。例如，在 Python 中，方括号 `[` 和 `]` 用于列表，花括号 `{` 和 `}` 用于字典。括号 `(` 和 `)` 用于元祖和算术表达式。只要每个符号都能保持自己的开始和结束关系，就可以混合符号。符号字符串如

```
{ { ( [ ] [ ] ) } ( ) }

[ [ { { ( ( ) ) } } ] ]

[ ] [ ] [ ] ( ) { }
```

这些被恰当的匹配了，因为不仅每个开始符号都有对应的结束符号，而且符号的类型也匹配。

相反这些字符串没法匹配：

```
( [ ) ]

( ( ( ) ] ) )

[ { ( ) ]
```

上节的简单括号检查程序可以轻松的扩展处理这些新类型的符号。回想一下，每个开始符号被简单的压入栈中，等待匹配的结束符号出现。当出现结束符号时，唯一的区别是我们必须检查确保它正确匹配栈顶部开始符号的类型。如果两个符号不匹配，则字符串不匹配。如果整个字符串都被处理完并且没有什么留在栈中，则字符串匹配。

Python 程序见 ActiveCode 1。唯一的变化是 16 行，我们称之为辅助函数匹配。必须检查栈中每个删除的符号，以查看它是否与当前结束符号匹配。如果不匹配，则布尔变量 balanced 被设置为 False。

```
from pythonds.basic.stack import Stack

def parChecker(symbolString):
    s = Stack()
    balanced = True
    index = 0
    while index < len(symbolString) and balanced:
        symbol = symbolString[index]
        if symbol in "([{":
            s.push(symbol)
        else:
            if s.isEmpty():
                balanced = False
            else:
                top = s.pop()
                if not matches(top,symbol):
                       balanced = False
        index = index + 1
    if balanced and s.isEmpty():
        return True
    else:
        return False

def matches(open,close):
    opens = "([{"
    closers = ")]}"
    return opens.index(open) == closers.index(close)


print(parChecker('{{([][])}()}'))
print(parChecker('[{()]'))
```

*ActiveCode 1*

这两个例子表明，栈是计算机语言结构处理非常重要的数据结构。几乎你能想到的任何嵌套符号必须按照平衡匹配的顺序。栈还有其他重要的用途，我们将在接下来的部分讨论。

# 3.8.十进制转换成二进制

在你学习计算机的过程中，你可能已经接触了二进制。二进制在计算机科学中是很重要的，因为存储在计算机内的所有值都是以 0 和 1 存储的。如果没有能力在二进制数和普通字符串之间转换，我们与计算机之间的交互非常棘手。

整数值是常见的数据项。他们一直用于计算机程序和计算。我们在数学课上学习它们，当然最后用十进制或者基数 10 来表示它们。十进制 233^10 以及对应的二进制表示 11101001^2 分别解释为

$$2\times10^2+3\times10^1+3\times10^0$$

$$1\times2^7+1\times2^6+1\times2^5+0\times2^4+1\times2^3+0\times2^2+0\times2^1+1\times2^0$$

但是我们如何能够容易地将整数值转换为二进制呢？答案是 “除 2”算法，它用栈来跟踪二进制结果的数字。

“除 2” 算法假定我们从大于 0 的整数开始。不断迭代的将十进制除以 2，并跟踪余数。第一个除以 2 的余数说明了这个值是偶数还是奇数。偶数有 0 的余数，记为 0。奇数有余数 1，记为 1.我们将得到的二进制构建为数字序列，第一个余数实际上是序列中的最后一个数字。见 Figure 5 , 我们再次看到了反转的属性，表示栈可能是解决这个问题的数据结构。

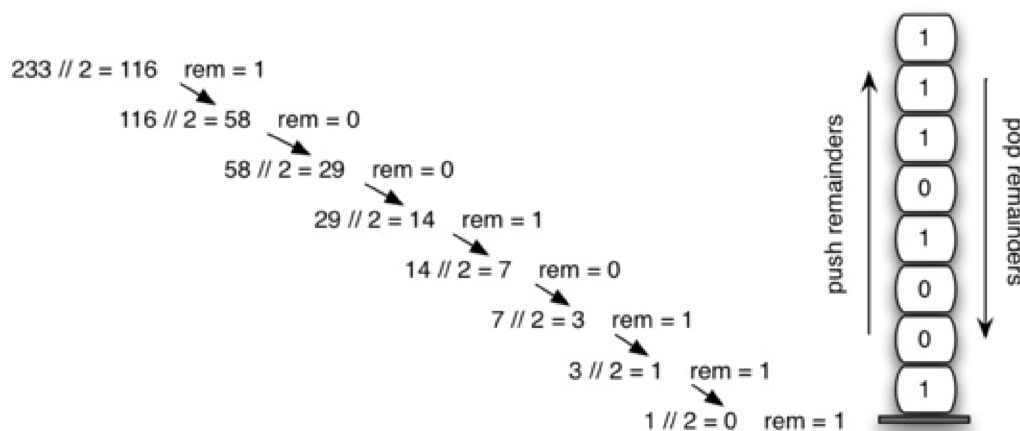



*Figure 5*

Activecode 1 中的 Python 代码实现了 “除 2” 算法，函数 divideBy2 传入了一个十进制的参数，并重复除以 2。第 7 行使用内置的模运算符 % 来提取余数，第 8 行将余数压到栈上。当除到 0 后，11-13 行构造了一个二进制字符串。

```
from pythonds.basic.stack import Stack

def divideBy2(decNumber):
    remstack = Stack()

    while decNumber > 0:
        rem = decNumber % 2
        remstack.push(rem)
        decNumber = decNumber // 2

    binString = ""
    while not remstack.isEmpty():
        binString = binString + str(remstack.pop())

    return binString

print(divideBy2(42))
```

*ActiveCode 1*

这个用于二进制转换的算法可以很容易的扩展以执行任何基数的转换。在计算机科学中，通常会使用很多不同的编码。其中最常见的是二级制，八进制和十六进制。

十进制 233 和它对应的八进制和十六进制 351^8, E9^16

$3 \times 8^2 + 5 \times 8^1 + 1 \times 8^0$ $14 \times 16^1 + 9 \times 16^0$

可以修改 divideBy2 函数，使它不仅能接受十进制参数，还能接受预期转换的基数。'除 2' 的概念被简单的替换成更通用的 '除基数'。在 ActiveCode2 展示的是一个名为 baseConverter 函数。采用十进制数和 2 到 16 之间的任何基数作为参数。余数部分仍然入栈，直到被转换的值为 0。我们创建一组数字，用来表示超过 9 的余数。

```
from pythonds.basic.stack import Stack

def baseConverter(decNumber,base):
    digits = "0123456789ABCDEF"

    remstack = Stack()

    while decNumber > 0:
        rem = decNumber % base
        remstack.push(rem)
        decNumber = decNumber // base

    newString = ""
    while not remstack.isEmpty():
        newString = newString + digits[remstack.pop()]

    return newString

print(baseConverter(25,2))
print(baseConverter(25,16))
```

*ActiveCode2*

# 3.9.中缀前缀和后缀表达式

当你编写一个算术表达式如 `B*C` 时，表达式的形式使你能够正确理解它。在这种情况下，你知道 B 乘以 C, 因为乘法运算符 `*` 出现在表达式中。这种类型的符号称为中缀，因为运算符在它处理的两个操作数之间。看另外一个中缀示例，`A+B*C`，运算符 `+` 和 `*` 仍然出现在操作数之间。这里面有个问题是，他们分别作用于哪个运算数上，`+` 作用于 A 和 B , 还是 `*` 作用于 B 和 C？表达式似乎有点模糊。

事实上，你已经读写过这些类型的表达式很长一段时间，所以它们对你不会导致什么问题。这是因为你知道运算符 `+` 和 `*`。每个运算符都有一个优先级。优先级较高的运算符在优先级较低的运算符之前使用。唯一改变顺序的是括号的存在。算术运算符的优先顺序是将乘法和除法置于加法和减法之上。如果出现具有相等优先级的两个运算符，则使用从左到右的顺序排序或关联。

我们使用运算符优先级来解释下表达式 `A+B*C`。B 和 C 首先相乘，然后将 A 与该结果相加。`(A+B)*C` 将强制在乘法之前执行 A 和 B 的加法。在表达式 `A+B+C` 中，最左边的 + 首先使用。

虽然这一切对你来说都很明显。但请记住，计算机需要准确知道要执行的操作符和顺序。一种保证不会对操作顺序产生混淆的表达式的方法是创建一个称为完全括号表达式的表达式。这种类型的表达式对每个运算符都使用一对括号。括号没有歧义的指示操作的顺序。也没有必要记住任何优先规则。

表达式 `A+B*C+D` 可以重写为 `((A + (B * C)) + D)` ，表明先乘法，然后是左边的加法，`A + B + C + D` 可以写为 `(((A + B) + C) + D)`，因为加法操作从左向右相关联。

有两种非常重要的表达式格式，你可能一开始不会很明显的看出来。中缀表达式 `A+B`, 如果我们移动两个操作数之间的运算符会发生什么？结果表达式变成 `+ A B`。同样，我们也可以将运算符移动到结尾，得到 `A B +` ，这样看起来有点奇怪。

改变操作符的位置得到了两种新的表达式格式，前缀和后缀。前缀表达式符号要求所有运算符在它们处理的两个操作数之前。另一个方面，后缀要求其操作符在相应的操作数之后。看下更多的例子 (见 Table 2)

`A+B*C` 将在前缀中写为 `+ A * B C` 。乘法运算符紧接在操作数 B 和 C 之前，表示 `*` 优先于 `+`。然后加法运算符出现在 A 和乘法的结果之前。

在后缀中，表达式将是 `A B C * +`，再次，操作的顺序被保留，因为 `*` 紧接在 B 和 C 之后出现，表示 `*` 具有高优先级，`+` 优先级低。虽然操作符在它们各自的操作数前后移动，但是顺序相对于彼此保持完全相同。

| Infix Expression | Prefix Expression | Postfix Expression |
|---|---|---|
| A + B | + A B | A B + |
| A + B * C | + A * B C | A B C * + |

*Table 2*

现在考虑中缀表达式 `(A + B) * C`，回想下，在这种情况下，中缀需要括号在乘法之前强制执行加法。然而，当 A+B 写到前缀中时，加法运算符简单的移动到操作数 `+ A B` 之前。这个操作的结果成为乘法的第一个操作数。乘法运算符移动到整个表达式的前面，得出 `* + A B C`，同样，在后缀 `A B +` 中，强制先加法。可以直接对该结果和剩余的操作数 C 相乘。然后，得出后缀表达式为 `A B + C *`。

再次考虑这三个表达式(见 Table 3)，括号不见了。为什么在前缀和后缀的时候不需要括号了呢？答案是操作符对于他们的操作数不再模糊，只有中缀才需要括号，前缀和后缀表达式的操作顺序完全由操作符的顺序决定。

| Infix Expression | Prefix Expression | Postfix Expression |
|---|---|---|
| (A + B) * C | * + A B C | A B + C * |

*Table 3*

Table 4 展示了一些其他的例子

| Infix Expression | Prefix Expression | Postfix Expression |
|---|---|---|
| A + B * C + D | + + A * B C D | A B C * + D + |
| (A + B) * (C + D) | * + A B + C D | A B + C D + * |
| A * B + C * D | + * A B * C D | A B * C D * + |
| A + B + C + D | + + + A B C D | A B + C + D + |

*Table 4*

## 3.9.1.中缀表达式转换前缀和后缀

到目前为止，我们已经使用特定方法在中缀表达式和等效前缀和后缀表达式符号之间进行转换。正如你可能期望的，有一些算法来执行转换，允许任何复杂表达式转换。

我们考虑的第一种技术使用前面讨论的完全括号表达式的概念。回想一下，`A + B * C` 可以写成 `(A +(B * C))`，以明确标识乘法优先于加法。然而，仔细观察,你可以看到每个括号对还表示操作数对的开始和结束，中间有相应的运算符。

看上面的子表达式 `(B * C)` 中的右括号。 如果我们将乘法符号移动到那个位置，并删除匹配的左括号，得到 `B C *` ，我们实际上已经将子表达式转换为后缀符号。 如果加法运算符也被移动到其相应的右括号位置并且匹配的左括号被去除，则将得到完整的后缀表达式（见 Figure 6）。




*Figure 6*

如果我们不是将符号移动到右括号的位置，我们将它向左移动，我们得到前缀符号（见 Figure 7）。圆括号对的位置实际上是包含的运算符的最终位置的线索。

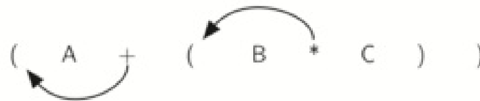



*Figure 7*

所以为了转换表达式，无论是对前缀还是后缀符号，先根据操作的顺序把表达式转换成完全括号表达式。然后将包含的运算符移动到左或右括号的位置，具体取决于需要前缀或后缀符号。

这里面有个更复杂的例子, `(A + B) * C - (D - E) * (F + G)` ，Figure 8 显示了如何转换为后缀和前缀。

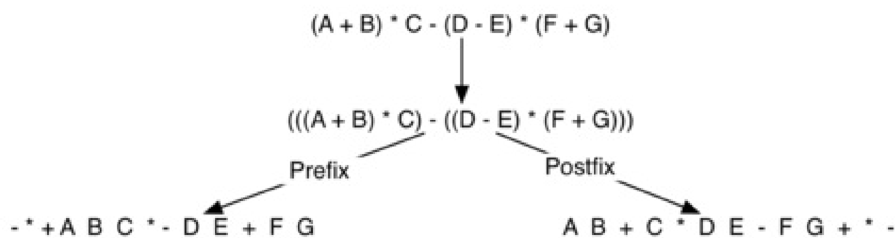



*Figure 8*

## 3.9.2.中缀转后缀通用法

我们需要开发一个算法来将任何中缀表达式转换为后缀表达式。 为了做到这一点，我们仔细看看转换过程。

再次考虑表达式 `A + B * C` 。如上所示，`A B C * +` 是等价的后缀表达式。 我们已经注意到，操作数 A，B 和 C 保持在它们的相对位置。只有操作符改变位置。再看中缀表达式中的运算符。从左到右出现的第一个运算符为 +。 然而，在后缀表达式中，+ 在结束位置，因为下一个运算符 * 的优先级高于加法。 原始表达式中的运算符的顺序在生成的后缀表达式中相反。

当我们处理表达式时，操作符必须保存在某处，因为它们相应的右操作数还没有看到。 此外，这些保存的操作符的顺序可能由于它们的优先级而需要反转。这是在该示例中的加法和乘法的情况，由于加法运算符在乘法运算符之前，并且具有较低的优先级，因此需要在使用乘法运算符之后出现。 由于这种顺序的反转，考虑使用栈来保存运算符直到用到它们是有意义的。

`(A + B)* C` 的情况会是什么样呢？ 回想一下，`A B + C *` 是等价的后缀表达式。从左到右处理此中缀表达式，我们先看到 `+` 。 在这种情况下，当我们看到 `*` ，`+` 已经放置在结果表达式中，由于括号它的优先级高于 `*` 。 我们现在可以开始看看转换算法如何工作。当我们看到左括号时，我们保存它，表示高优先级的另一个运算符将出现。该操作符需要等到相应的右括号出现以表示其位置（回忆完全括号的算法）。 当右括号出现时，可以从栈中弹出操作符。

当我们从左到右扫描中缀表达式时，我们将使用栈来保留运算符。这将提供我们在第一个例子中注意到的反转。 堆栈的顶部将始终是最近保存的运算符。每当我们读取一个新的运算符时，我们需要考虑该运算符如何与已经在栈上的运算符（如果有的话）比较优先级。

假设中缀表达式是一个由空格分隔的标记字符串。 操作符标记是 `*，/，+` 和 `-` ，以及左右括号。操作数是单字符 A，B，C 等。 以下步骤将后缀顺序生成一个字符串。

1. 创建一个名为 opstack 的空栈以保存运算符。给输出创建一个空列表。
2. 通过使用字符串方法拆分将输入的中缀字符串转换为标记列表。
3. 从左到右扫描标记列表。
    - 如果标记是操作数，将其附加到输出列表的末尾。
    - 如果标记是左括号，将其压到 opstack 上。
    - 如果标记是右括号，则弹出 opstack，直到删除相应的左括号。将每个运算符附加到输出列表的末尾。
    - 如果标记是运算符，`*，/，+` 或 `-` ，将其压入 opstack。但是，首先删除已经在 opstack 中具有更高或相等优先级的任何运算符，并将它们加到输出列表中。
4. 当输入表达式被完全处理时，检查 opstack。仍然在栈上的任何运算符都可以删除并加到输出列表的末尾。

Figure 9 展示了对表达式 `A * B + C * D` 的转换算法。注意，第一个 `*` 在看到 `+` 运算符时被删除。另外，当第二个 * 出现时， `+` 保留在栈中，因为乘法优先级高于加法。在中缀表达式的末尾，栈被弹出两次，删除两个运算符，并将 `+` 作为后缀表达式中的最后一个运算符。

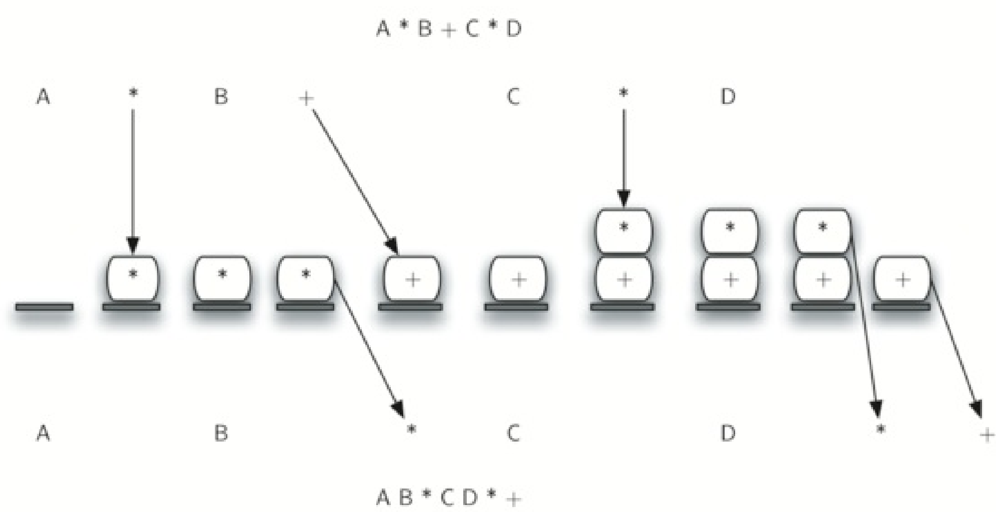



*Figure 9*

为了在 Python 中编写算法，我们使用一个名为 prec 的字典来保存操作符的优先级。这个字典将每个运算符映射到一个整数，可以与其他运算符的优先级（我们使用整数3，2和1）进行比较。左括号将赋予最低的值。这样，与其进行比较的任何运算符将具有更高的优先级，将被放置在它的顶部。第15行将操作数定义为任何大写字符或数字。完整的转换函数见 ActiveCode 1。

```
from pythonds.basic.stack import Stack

def infixToPostfix(infixexpr):
    prec = {}
    prec["*"] = 3
    prec["/"] = 3
    prec["+"] = 2
    prec["-"] = 2
    prec["("] = 1
    opStack = Stack()
    postfixList = []
    tokenList = infixexpr.split()

    for token in tokenList:
        if token in "ABCDEFGHIJKLMNOPQRSTUVWXYZ" or token in "0123456789":
            postfixList.append(token)
        elif token == '(':
            opStack.push(token)
        elif token == ')':
            topToken = opStack.pop()
            while topToken != '(':
                postfixList.append(topToken)
                topToken = opStack.pop()
        else:
            while (not opStack.isEmpty()) and \
               (prec[opStack.peek()] >= prec[token]):
                  postfixList.append(opStack.pop())
            opStack.push(token)

    while not opStack.isEmpty():
        postfixList.append(opStack.pop())
    return " ".join(postfixList)

print(infixToPostfix("A * B + C * D"))
print(infixToPostfix("( A + B ) * C - ( D - E ) * ( F + G )"))
```

执行结果如下

```
>>> infixtopostfix("( A + B ) * ( C + D )")
'A B + C D + *'
>>> infixtopostfix("( A + B ) * C")
'A B + C *'
>>> infixtopostfix("A + B * C")
'A B C * +'
>>>
```

## 3.9.3.后缀表达式求值

作为最后栈的示例，我们考虑对后缀符号中的表达式求值。在这种情况下，栈再次是我们选择的数据结构。但是，在扫描后缀表达式时，它必须等待操作数，而不像上面的转换算法中的运算符。 解决问题的另一种方法是，每当在输入上看到运算符时，计算两个最近的操作数。

要详细的了解这一点，考虑后缀表达式 `4 5 6 * +`， 首先遇到操作数 `4` 和 `5`，此时，你还不确定如何处理它们，直到看到下一个符号。将它们放置到栈上，确保它们在下一个操作符出现时可用。

在这种情况下，下一个符号是另一个操作数。所以，像先前一样，压入栈中。并检查下一个符号。现在我们看到了操作符 `*`，这意味着需要将两个最近的操作数相乘。通过弹出栈两次，我们可以得到正确的两个操作数，然后执行乘法（这种情况下结果为 30）。

我们现在可以通过将其放回栈中来处理此结果，以便它可以表示为表达式后面的运算符的操作数。当处理最后一个操作符时，栈上只有一个值，弹出并返回它作为表达式的结果。Figure 10 展示了整个示例表达式的栈的内容。

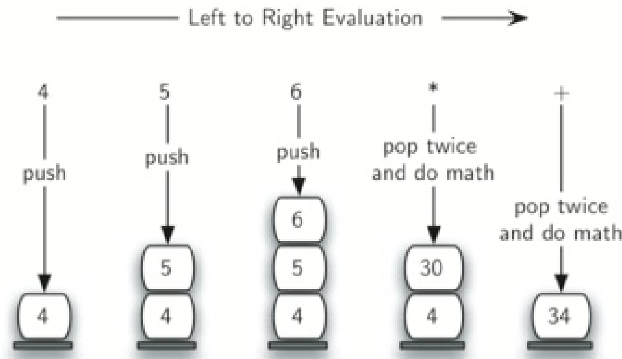



*Figure 10*

Figure 11 是个稍微复杂的示例，`7 8 + 3 2 + /` 。在这个例子中有两点需要注意，首先，栈的大小增长收缩，然后再子表达式求值的时候再次增长。第二，除法操作需要自信处理。回想下，后缀表达式的操作符顺序没变，仅仅改变操作符的位置。当用于除法的操作符从栈中弹出时，它们被反转。由于除法不是交换运算符，换句话说 `15/5` 和 `5/15` 不同，因此我们必须保证操作数的顺序不会交换。

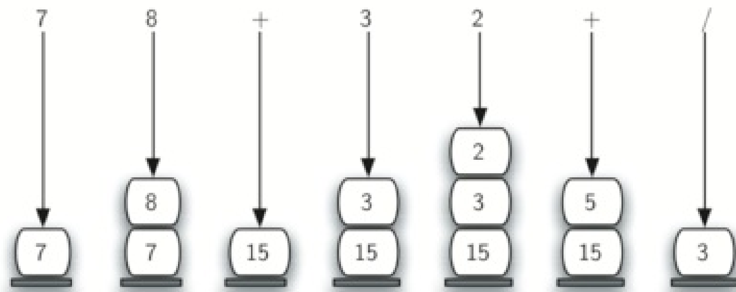


*Figure 11*

假设后缀表达式是一个由空格分隔的标记字符串。 运算符为 `*，/，+` 和 `-` ，操作数假定为单个整数值。 输出将是一个整数结果。

1. 创建一个名为 `operandStack` 的空栈。
2. 拆分字符串转换为标记列表。
3. 从左到右扫描标记列表。
   - 如果标记是操作数，将其从字符串转换为整数，并将值压到operandStack。
   - 如果标记是运算符 `*，/，+` 或 `-` ，它将需要两个操作数。弹出operandStack 两次。第一个弹出的是第二个操作数，第二个弹出的是第一个操作数。执行算术运算后，将结果压到操作数栈中。
4. 当输入的表达式被完全处理后，结果就在栈上，弹出 operandStack 并返回值。

用于计算后缀表达式的完整函数见 ActiveCode 2，为了辅助计算，定义了一个函数 doMath,它将获取两个操作数和运算符，执行相应的计算。

```
from pythonds.basic.stack import Stack

def postfixEval(postfixExpr):
    operandStack = Stack()
    tokenList = postfixExpr.split()

    for token in tokenList:
        if token in "0123456789":
            operandStack.push(int(token))
        else:
            operand2 = operandStack.pop()
            operand1 = operandStack.pop()
            result = doMath(token,operand1,operand2)
            operandStack.push(result)
    return operandStack.pop()

def doMath(op, op1, op2):
    if op == "*":
        return op1 * op2
    elif op == "/":
        return op1 / op2
    elif op == "+":
        return op1 + op2
    else:
        return op1 - op2

print(postfixEval('7 8 + 3 2 + /'))
```

# 3.10.什么是队列

队列是项的有序结合，其中添加新项的一端称为队尾，移除项的一端称为队首。当一个元素从队尾进入队列时，一直向队首移动，直到它成为下一个需要移除的元素为止。

最近添加的元素必须在队尾等待。集合中存活时间最长的元素在队首，这种排序成为 FIFO，先进先出，也被成为先到先得。

队列的最简单的例子是我们平时不时会参与的列。排队等待电影，在杂货店的收营台等待，在自助餐厅排队等待（这样我们可以弹出托盘栈）。行为良好的线或队列是有限制的，因为它只有一条路，只有一条出路。不能插队，也不能离开。你只有等待了一定的时间才能到前面。Figure 1 展示了一个简单的 Python 对象队列。




*Figure 1*

计算机科学也有常见的队列示例。我们的计算机实验室有 30 台计算机与一台打印机联网。当学生想要打印时，他们的打印任务与正在等待的所有其他打印任务“一致”。第一个进入的任务是先完成。如果你是最后一个，你必须等待你前面的所有其他任务打印。我们将在后面更详细地探讨这个有趣的例子。

除了打印队列，操作系统使用多个不同的队列来控制计算机内的进程。下一步做什么的调度通常基于尽可能快地执行程序和尽可能多的服务用户的排队算法。此外，当我们敲击键盘时，有时屏幕上出现的字符会延迟。这是由于计算机在那一刻做其他工作。按键的内容被放置在类似队列的缓冲器中，使得它们最终可以以正确的顺序显示在屏幕上。

# 3.11.队列抽象数据类型

队列抽象数据类型由以下结构和操作定义。如上所述，队列被构造为在队尾添加项的有序集合，并且从队首移除。队列保持 FIFO 排序属性。 队列操作如下。

- Queue() 创建一个空的新队列。 它不需要参数，并返回一个空队列。
- enqueue(item) 将新项添加到队尾。 它需要 item 作为参数，并不返回任何内容。
- dequeue() 从队首移除项。它不需要参数并返回 item。 队列被修改。
- isEmpty() 查看队列是否为空。它不需要参数，并返回布尔值。
- size() 返回队列中的项数。它不需要参数，并返回一个整数。

作为示例，我们假设 q 是已经创建并且当前为空的队列，则 Table 1 展示了队列操作序列的结果。右边表示队首。 4 是第一个入队的项，因此它 dequeue 返回的第一个项。

| Queue Operation | Queue Contents | Return Value |
|---|---|---|
| `q.isEmpty()` | `[]` | `True` |
| `q.enqueue(4)` | `[4]` | |
| `q.enqueue('dog')` | `['dog',4]` | |
| `q.enqueue(True)` | `[True,'dog',4]` | |
| `q.size()` | `[True,'dog',4]` | `3` |
| `q.isEmpty()` | `[True,'dog',4]` | `False` |
| `q.enqueue(8.4)` | `[8.4,True,'dog',4]` | |
| `q.dequeue()` | `[8.4,True,'dog']` | `4` |
| `q.dequeue()` | `[8.4,True]` | `'dog'` |
| `q.size()` | `[8.4,True]` | `2` |

*Table 1*

# 3.12.Python实现队列

我们为了实现队列抽象数据类型创建一个新类。和前面一样，我们将使用列表集合来作为构建队列的内部表示。

我们需要确定列表的哪一端作为队首，哪一端作为队尾。Listing 1 所示的实现假定队尾在列表中的位置为 0。这允许我们使用列表上的插入函数向队尾添加新元素。弹出操作可用于删除队首的元素（列表的最后一个元素）。回想一下，这也意味着入队为 O(n)，出队为 O(1)。

```
class Queue:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0,item)

    def dequeue(self):
        return self.items.pop()

    def size(self):
        return len(self.items)
```

*Listing 1*

进一步的操作这个队列产生如下结果：

```
>>> q.size()
3
>>> q.isEmpty()
False
>>> q.enqueue(8.4)
>>> q.dequeue()
4
>>> q.dequeue()
'dog'
>>> q.size()
2
```

# 3.13.模拟：烫手山芋

队列的典型应用之一是模拟需要以 FIFO 方式管理数据的真实场景。首先，让我们看看孩子们的游戏烫手山芋，在这个游戏中（见 Figure 2），孩子们围成一个圈，并尽可能快的将一个山芋递给旁边的孩子。在某一个时间，动作结束，有山芋的孩子从圈中移除。游戏继续开始直到剩下最后一个孩子。

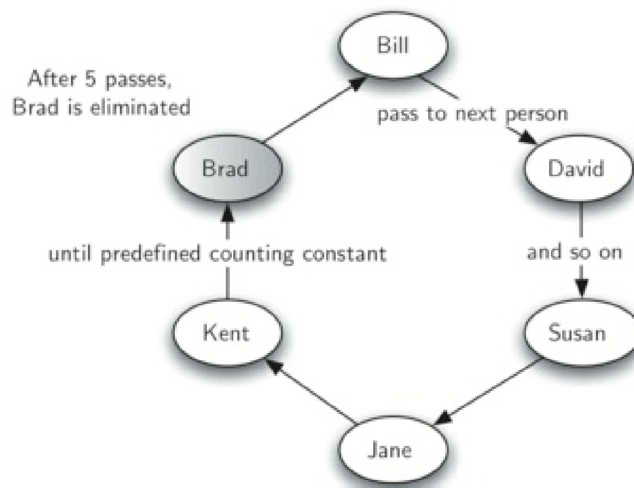



*Figure 2*

这个游戏相当于著名的约瑟夫问题，一个一世纪著名历史学家弗拉维奥·约瑟夫斯的传奇故事。故事讲的是，他和他的 39 个战友被罗马军队包围在洞中。他们决定宁愿死，也不成为罗马人的奴隶。他们围成一个圈，其中一人被指定为第一个人，顺时针报数到第七人，就将他杀死。约瑟夫斯是一个成功的数学家，他立即想出了应该坐到哪才能成为最后一人。最后，他加入了罗马的一方，而不是杀了自己。你可以找到这个故事的不同版本，有些说是每次报数 3 个人，有人说允许最后一个人逃跑。无论如何，思想是一样的。

我们将模拟这个烫山芋的过程。我们的程序将输入名称列表和一个称为 num 常量用于报数。它将返回以 `num` 为单位重复报数后剩余的最后一个人的姓名。

为了模拟这个圈，我们使用队列（见 Figure3）。假设拿着山芋的孩子在队列的前面。当拿到山芋的时候，这个孩子将先出列再入队列，把他放在队列的最后。经过 num 次的出队入队后，前面的孩子将被永久移除队列。并且另一个周期开始，继续此过程，直到只剩下一个名字（队列的大小为 1）。

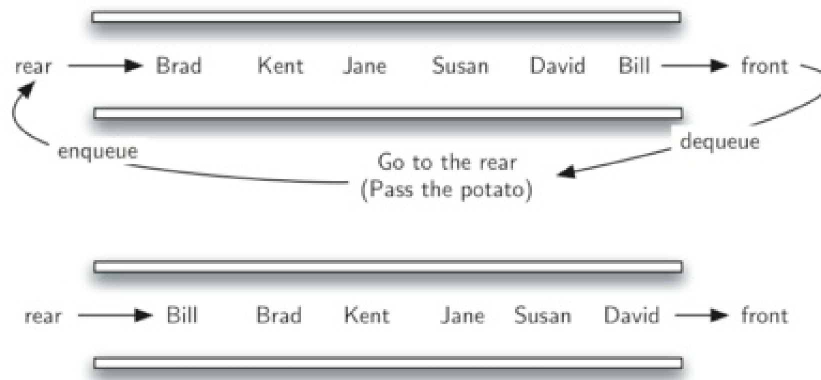



*Figure 3*

```
from pythonds.basic.queue import Queue

def hotPotato(namelist, num):
    simqueue = Queue()
    for name in namelist:
        simqueue.enqueue(name)

    while simqueue.size() > 1:
        for i in range(num):
            simqueue.enqueue(simqueue.dequeue())

        simqueue.dequeue()

    return simqueue.dequeue()

print(hotPotato(["Bill","David","Susan","Jane","Kent","Brad"],7))
```

*Active code 1*

请注意，在此示例中，计数常数的值大于列表中的名称数。这不是一个问题，因为队列像一个圈，计数会重新回到开始，直到达到计数值。另外，请注意，列表加载到队列中以使列表上的名字位于队列的前面。在这种情况下，`Bill` 是列表中的第一个项，因此他在队列的前面。

# 3.14.模拟：打印机

一个更有趣的模拟是允许我们研究本节前面描述的打印机的行为，回想一下，当学生向共享打印机发送打印任务时，任务被放置在队列中以便以先来先服务的方式被处理。此配置会出现许多问题。其中最重要的点可能是打印机是否能够处理一定量的工作。如果它不能，学生将等待太长时间打印，可能会错过他们的下一节课。

在计算机科学实验室里考虑下面的情况。平均每天大约10名学生在任何给定时间在实验室工作。这些学生通常在此期间打印两次，这些任务的长度范围从1到20页。实验室中的打印机较旧，每分钟以草稿质量可以处理10页。打印机可以切换以提供更好的质量，但是它将每分钟只能处理五页。较慢的打印速度可能会使学生等待太久。应使用什么页面速率？

我们可以通过建立一个模拟实验来决定。我们将需要为学生，打印任务和打印机构建表现表示（Figure 4）。当学生提交打印任务时，我们将把他们添加到等待列表中，一个打印任务的队列。 当打印机完成任务时，它将检查队列，以检查是否有剩余的任务要处理。我们感兴趣的是学生等待他们的论文打印的平均时间。这等于任务在队列中等待的平均时间量。

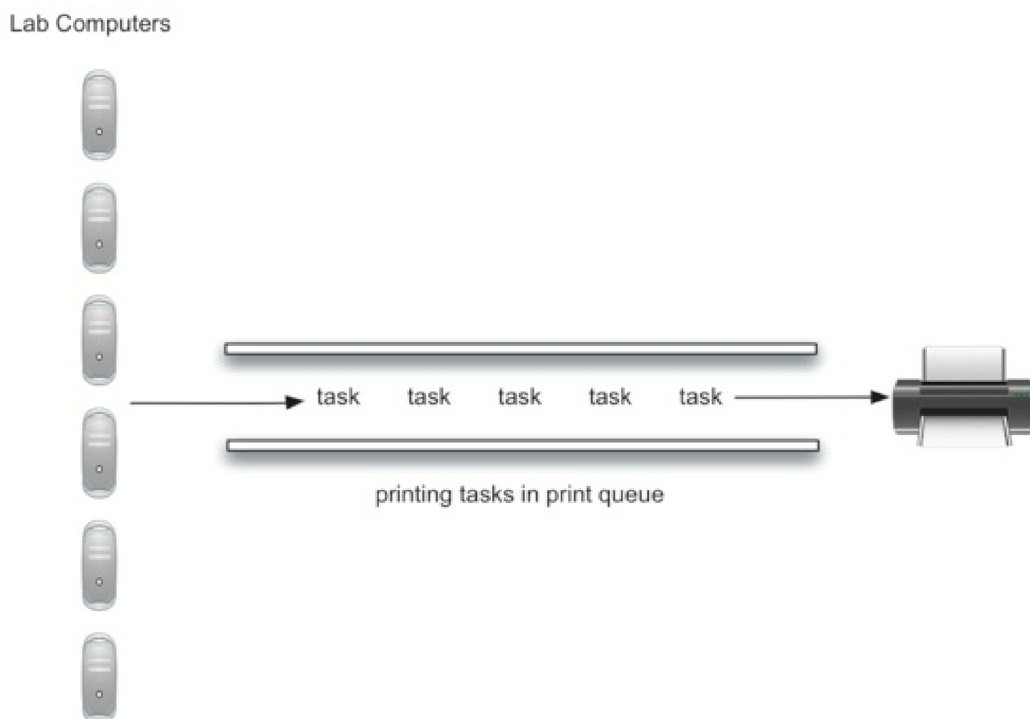



*Figure 4*

为了为这种情况建模，我们需要使用一些概率。例如，学生可以打印长度从 1 到 20 页的纸张。如果从 1 到 20 的每个长度有同样的可能性，则可以通过使用 1 和 20 之间的随机数来模拟打印任务的实际长度。这意味着出现从 1 到 20 的任何长度的机会是平等的。

如果实验室中有 10 个学生，每人打印两次，则平均每小时有 20 个打印任务。 在任何给定的秒，打印任务将被创建的机会是什么？ 回答这个问题的方法是考虑任务与时间的比率。每小时 20 个任务意味着平均每 180 秒将有一个任务：

$$\frac{20\ tasks}{1\ hour} \times \frac{1\ hour}{60\ minutes} \times \frac{1\ minute}{60\ seconds} = \frac{1\ task}{180\ seconds}$$

对于每一秒，我们可以通过生成 1 到 180 之间的随机数来模拟打印任务发生的机会。如果数字是 180，我们说一个任务已经创建。请注意，可能会在一下子创建许多任务，或者需要等待一段时间才有任务。这就是模拟的本质。你想模拟真实的情况就需要尽可能接近一般参数。

## 3.14.1.主要模拟步骤

1. 创建打印任务的队列，每个任务都有个时间戳。队列启动的时候为空。
2. 每秒（currentSecond）：
    - 是否创建新的打印任务？如果是，将 `currentSecond` 作为时间戳添加到队列。
    - 如果打印机不忙并且有任务在等待
        - 从打印机队列中删除一个任务并将其分配给打印机
        - 从 `currentSecond` 中减去时间戳，以计算该任务的等待时间。
        - 将该任务的等待时间附件到列表中稍后处理。
        - 根据打印任务的页数，确定需要多少时间。
    - 打印机需要一秒打印，所以得从该任务的所需的等待时间减去一秒。
    - 如果任务已经完成，换句话说，所需的时间已经达到零，打印机空闲。
3. 模拟完成后，从生成的等待时间列表中计算平均等待时间。

## 3.14.2 Python 实现

为了设计此模拟，我们将为上述三个真实世界对象创建类：`Printer`，`Task`，`PrintQueue`

`Printer` 类（Listing 2）需要跟踪它当前是否有任务。如果有，则它处于忙碌状态（13-17 行），并且可以从任务的页数计算所需的时间。构造函数允许初始化每分钟页面的配置，`tick` 方法将内部定时器递减直到打印机设置为空闲(11 行)

```
class Printer:
    def __init__(self, ppm):
        self.pagerate = ppm
        self.currentTask = None
        self.timeRemaining = 0

    def tick(self):
        if self.currentTask != None:
            self.timeRemaining = self.timeRemaining - 1
            if self.timeRemaining <= 0:
                self.currentTask = None

    def busy(self):
        if self.currentTask != None:
            return True
        else:
            return False

    def startNext(self,newtask):
        self.currentTask = newtask
        self.timeRemaining = newtask.getPages() * 60/self.pagerate
```

*Listing 2*

`Task` 类（Listing 3）表示单个打印任务。创建任务时，随机数生成器将提供 1 到 20 页的长度。我们选择使用随机模块中的 `randrange` 函数。

```
>>> import random
>>> random.randrange(1,21)
18
>>> random.randrange(1,21)
8
>>>
```

每个任务还需要保存一个时间戳用于计算等待时间。此时间戳将表示任务被创建并放置到打印机队列中的时间。可以使用 `waitTime` 方法来检索在打印开始之前队列中花费的时间。

```
import random

class Task:
    def __init__(self,time):
        self.timestamp = time
        self.pages = random.randrange(1,21)

    def getStamp(self):
        return self.timestamp

    def getPages(self):
        return self.pages

    def waitTime(self, currenttime):
        return currenttime - self.timestamp
```

*Listing 3*

Listing 4 实现了上述算法。`PrintQueue` 对象是我们现有队列 ADT 的一个实例。`newPrintTask` 决定是否创建一个新的打印任务。我们再次选择使用随机模块的 `randrange` 函数返回 1 到 180 之间的随机整数。打印任务每 180 秒到达一次。通过从随机整数（32 行）的范围中任意选择，我们可以模拟这个随机事件。模拟功能允许我们设置打印机的总时间和每分钟的页数。

```
from pythonds.basic.queue import Queue

import random

def simulation(numSeconds, pagesPerMinute):

    labprinter = Printer(pagesPerMinute)
    printQueue = Queue()
    waitingtimes = []

    for currentSecond in range(numSeconds):

      if newPrintTask():
         task = Task(currentSecond)
         printQueue.enqueue(task)

      if (not labprinter.busy()) and (not printQueue.isEmpty()):
        nexttask = printQueue.dequeue()
        waitingtimes.append(nexttask.waitTime(currentSecond))
        labprinter.startNext(nexttask)

      labprinter.tick()

    averageWait=sum(waitingtimes)/len(waitingtimes)
    print("Average Wait %6.2f secs %3d tasks remaining."%(averageWait,printQueue.size(
)))

def newPrintTask():
    num = random.randrange(1,181)
    if num == 180:
        return True
    else:
        return False

for i in range(10):
    simulation(3600,5)
```

*Listing 4*

当我们运行模拟时，我们不应该担心每次的结果不同。这是由于随机数的概率性质决定的。因为模拟的参数可以被调整，我们对调整后可能发生的趋势感兴趣。 这里有一些结果。

首先，我们将使用每分钟五页的页面速率运行模拟 60 分钟（3,600秒）。 此外，我们将进行 10 次独立试验。记住，因为模拟使用随机数，每次运行将返回不同的结果。

```
>>>for i in range(10):
      simulation(3600,5)

Average Wait 165.38 secs 2 tasks remaining.
Average Wait  95.07 secs 1 tasks remaining.
Average Wait  65.05 secs 2 tasks remaining.
Average Wait  99.74 secs 1 tasks remaining.
Average Wait  17.27 secs 0 tasks remaining.
Average Wait 239.61 secs 5 tasks remaining.
Average Wait  75.11 secs 1 tasks remaining.
Average Wait  48.33 secs 0 tasks remaining.
Average Wait  39.31 secs 3 tasks remaining.
Average Wait 376.05 secs 1 tasks remaining.
```

在运行 10 次实验后，我们可以看到，平均等待时间为 122.09 秒。 还可以看到平均等待时间有很大的变化，最小值为 17.27 秒，最大值为 376.05 秒。 你也可能注意到，只有两种情况所有任务都完成。

现在，我们将页面速率调整为每分钟 10 页，再次运行 10 次测试，页面速度更快，我们希望在一小时内完成更多的任务。

```
>>>for i in range(10):
      simulation(3600,10)

Average Wait   1.29 secs 0 tasks remaining.
Average Wait   7.00 secs 0 tasks remaining.
Average Wait  28.96 secs 1 tasks remaining.
Average Wait  13.55 secs 0 tasks remaining.
Average Wait  12.67 secs 0 tasks remaining.
Average Wait   6.46 secs 0 tasks remaining.
Average Wait  22.33 secs 0 tasks remaining.
Average Wait  12.39 secs 0 tasks remaining.
Average Wait   7.27 secs 0 tasks remaining.
Average Wait  18.17 secs 0 tasks remaining.
```

## 3.14.3.讨论

我们试图回答一个问题，即当前打印机是否可以处理任务负载，如果它设置为打印更好的质量，较慢的页面速率。我们采用的方法是编写一个模拟打印任务作为各种页数和到达时间的随机事件的模拟。

上面的输出显示，每分钟打印 5 页，平均等待时间从低的 17 秒到高的 376 秒（约 6 分钟）。使用更快的打印速率，低值为 1 秒，高值仅为 28。此外，在 10 次运行中的 8 次，每分钟 5 页，打印任务在结束时仍在队列中等待。

因此，我们说减慢打印机的速度以获得更好的质量可能不是一个好主意。学生们不能等待他们的论文打印完，特别是当他们需要到下一个班级。六分钟的等待时间太长了。

这种类型的模拟分析允许我们回答许多问题，通常被称为“如果”的问题。我们需要做的是改变模拟使用的参数，我们可以模拟任何数量。例如

```
* 如果入学人数增加，平均学生人数增加 20 人 该怎么办？
* 如果是星期六，学生不需要上课怎么办？他们能负担得了吗？
* 如果平均打印任务的大小减少了，由于 Python 是一个强大的语言，程序往往要短得多？
```

这些问题都可以通过修改上述模拟来回答。然而，重要的是要记住，模拟有效取决于构建它的假设是没问题的。关于每小时打印任务的数量和每小时的学生数量的真实数据对于构建鲁棒性的模拟是必要的。

# 3.15.什么是Deque

deque（也称为双端队列）是与队列类似的项的有序集合。它有两个端部，首部和尾部，并且项在集合中保持不变。deque 不同的地方是添加和删除项是非限制性的。可以在前面或后面添加新项。同样，可以从任一端移除现有项。在某种意义上，这种混合线性结构提供了单个数据结构中的栈和队列的所有能力。 Figure 1 展示了一个 Python 数据对象的 deque 。

要注意，即使 deque 可以拥有栈和队列的许多特性，它不需要由那些数据结构强制的 LIFO 和 FIFO 排序。这取决于你如何持续添加和删除操作。

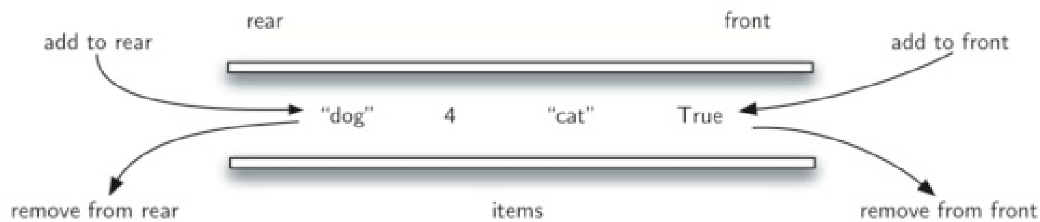



*Figure 1*

# 3.16.Deque抽象数据类型

deque 抽象数据类型由以下结构和操作定义。如上所述，deque 被构造为项的有序集合，其中项从首部或尾部的任一端添加和移除。下面给出了 deque 操作。

- Deque() 创建一个空的新 deque。它不需要参数，并返回空的 deque。
- addFront(item) 将一个新项添加到 deque 的首部。它需要 item 参数 并不返回任何内容。
- addRear(item) 将一个新项添加到 deque 的尾部。它需要 item 参数并不返回任何内容。
- removeFront() 从 deque 中删除首项。它不需要参数并返回 item。deque 被修改。
- removeRear() 从 deque 中删除尾项。它不需要参数并返回 item。deque 被修改。
- isEmpty() 测试 deque 是否为空。它不需要参数，并返回布尔值。
- size() 返回 deque 中的项数。它不需要参数，并返回一个整数。

例如，我们假设 d 是已经创建并且当前为空的 deque，则 Table 1 展示了一系列 deque 操作的结果。注意，首部的内容列在右边。在将 item 移入和移出时，跟踪前面和后面是非常重要的，因为可能会有点混乱。

| Deque Operation | Deque Contents | Return Value |
|---|---|---|
| `d.isEmpty()` | `[]` | `True` |
| `d.addRear(4)` | `[4]` | |
| `d.addRear('dog')` | `['dog',4,]` | |
| `d.addFront('cat')` | `['dog',4,'cat']` | |
| `d.addFront(True)` | `['dog',4,'cat',True]` | |
| `d.size()` | `['dog',4,'cat',True]` | `4` |
| `d.isEmpty()` | `['dog',4,'cat',True]` | `False` |
| `d.addRear(8.4)` | `[8.4,'dog',4,'cat',True]` | |
| `d.removeRear()` | `['dog',4,'cat',True]` | `8.4` |
| `d.removeFront()` | `['dog',4,'cat']` | `True` |

*Table 1*

# 3.17.Python实现Deque

正如我们在前面的部分中所做的，我们将为抽象数据类型 deque 的实现创建一个新类。同样，Python 列表将提供一组非常好的方法来构建 deque 的细节。我们的实现（Listing 1）将假定 deque 的尾部在列表中的位置为 0。

```
class Deque:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def addFront(self, item):
        self.items.append(item)

    def addRear(self, item):
        self.items.insert(0,item)

    def removeFront(self):
        return self.items.pop()

    def removeRear(self):
        return self.items.pop(0)

    def size(self):
        return len(self.items)
```

*Listing 1*

在 removeFront 中，我们使用 pop 方法从列表中删除最后一个元素。 但是，在removeRear 中，pop(0)方法必须删除列表的第一个元素。同样，我们需要在 addRear 中使用insert方法（第12行），因为 append 方法在列表的末尾添加一个新元素。

你可以看到许多与栈和队列中描述的 Python 代码相似之处。你也可能观察到，在这个实现中，从前面添加和删除项是 O(1)，而从后面添加和删除是 O(n)。 考虑到添加和删除项是出现的常见操作，这是可预期的。 同样，重要的是要确定我们知道在实现中前后都分配在哪里。

# 3.18.回文检查

使用 deque 数据结构可以容易地解决经典回文问题。回文是一个字符串，读取首尾相同的字符，例如，`radar toot madam`。我们想构造一个算法输入一个字符串，并检查它是否是一个回文。

该问题的解决方案将使用 deque 来存储字符串的字符。我们从左到右处理字符串，并将每个字符添加到 deque 的尾部。在这一点上，deque 像一个普通的队列。然而，我们现在可以利用 deque 的双重功能。deque 的首部保存字符串的第一个字符，deque 的尾部保存最后一个字符（见 Figure 2）。

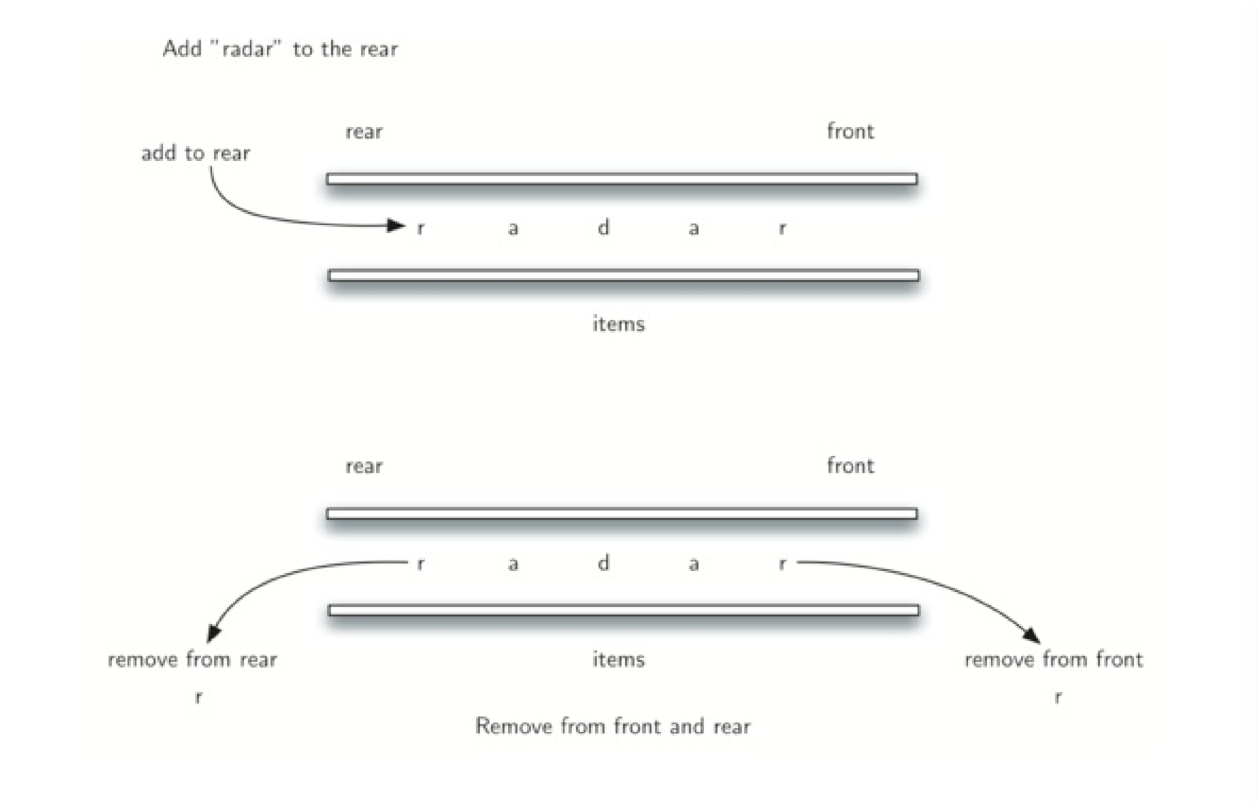



*Figure 2*

我们可以直接删除并比较首尾字符，只有当它们匹配时才继续。如果可以持续匹配首尾字符，我们最终要么用完字符，要么留出大小为 1 的deque，取决于原始字符串的长度是偶数还是奇数。在任一情况下，字符串都是回文。回文检查的完整功能在 ActiveCode 1 中。

```
from pythonds.basic.deque import Deque

def palchecker(aString):
    chardeque = Deque()

    for ch in aString:
        chardeque.addRear(ch)

    stillEqual = True

    while chardeque.size() > 1 and stillEqual:
        first = chardeque.removeFront()
        last = chardeque.removeRear()
        if first != last:
            stillEqual = False

    return stillEqual

print(palchecker("lsdkjfskf"))
print(palchecker("radar"))
```

*ActiveCode 1*

# 3.19.列表

在对基本数据结构的讨论中，我们使用 Python 列表来实现所呈现的抽象数据类型。列表是一个强大但简单的收集机制，为程序员提供了各种各样的操作。然而，不是所有的编程语言都包括列表集合。在这些情况下，列表的概念必须由程序员实现。

列表是项的集合，其中每个项保持相对于其他项的相对位置。更具体地，我们将这种类型的列表称为无序列表。我们可以将列表视为具有第一项，第二项，第三项等等。我们还可以引用列表的开头（第一个项）或列表的结尾（最后一个项）。为了简单起见，我们假设列表不能包含重复项。

例如，整数 `54,26,93,17,77` 和 `31` 的集合可以表示考试分数的简单无序列表。请注意，我们将它们用逗号分隔，这是列表结构的常用方式。当然，Python 会显示这个列表为 `[54,26,93,17,77,31]` 。

# 3.20.无序列表抽象数据类型

如上所述，无序列表的结构是项的集合，其中每个项保持相对于其他项的相对位置。下面给出了一些可能的无序列表操作。

- List() 创建一个新的空列表。它不需要参数，并返回一个空列表。
- add(item) 向列表中添加一个新项。它需要 item 作为参数，并不返回任何内容。假定该 item 不在列表中。
- remove(item) 从列表中删除该项。它需要 item 作为参数并修改列表。假设项存在于列表中。
- search(item) 搜索列表中的项目。它需要 item 作为参数，并返回一个布尔值。
- isEmpty() 检查列表是否为空。它不需要参数，并返回布尔值。
- size（）返回列表中的项数。它不需要参数，并返回一个整数。
- append(item) 将一个新项添加到列表的末尾，使其成为集合中的最后一项。它需要 item 作为参数，并不返回任何内容。假定该项不在列表中。
- index(item) 返回项在列表中的位置。它需要 item 作为参数并返回索引。假定该项在列表中。
- insert(pos，item) 在位置 pos 处向列表中添加一个新项。它需要 item 作为参数并不返回任何内容。假设该项不在列表中，并且有足够的现有项使其有 pos 的位置。
- pop() 删除并返回列表中的最后一个项。假设该列表至少有一个项。
- pop(pos) 删除并返回位置 pos 处的项。它需要 pos 作为参数并返回项。假定该项在列表中。

# 3.21.实现无序列表：链表

为了实现无序列表，我们将构造通常所知的链表。回想一下，我们需要确保我们可以保持项的相对定位。然而，没有要求我们维持在连续存储器中的定位。例如，考虑 Figure 1 中所示的项的集合。看来这些值已被随机放置。如果我们可以在每个项中保持一些明确的信息，即下一个项的位置（参见 Figure 2），则每个项的相对位置可以通过简单地从一个项到下一个项的链接来表示。




*Figure 1*

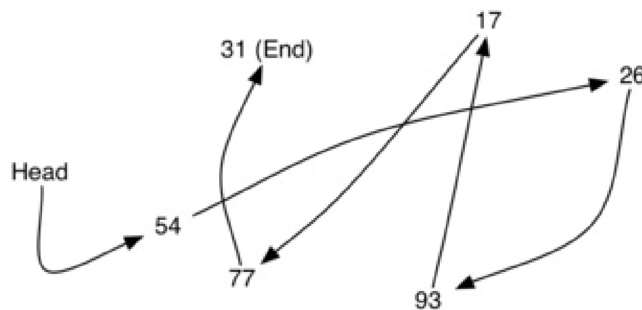



*Figure 2*

要注意，必须明确地指定链表的第一项的位置。一旦我们知道第一个项在哪里，第一个项目可以告诉我们第二个是什么，等等。外部引用通常被称为链表的头。类似地，最后一个项需要知道没有下一个项。

## 3.21.1.Node 类

链表实现的基本构造块是节点。每个节点对象必须至少保存两个信息。首先，节点必须包含列表项本身。我们将这个称为节点的数据字段。此外，每个节点必须保存对下一个节点的引用。 Listing 1 展示了 Python 实现。要构造一个节点，需要提供该节点的初始数据值。下面

的赋值语句将产生一个包含值 93 的节点对象（见 Figure 3）。应该注意，我们通常会如 Figure 4 所示表示一个节点对象。Node 类还包括访问，修改数据和访问下一个引用的常用方法。

```
class Node:
    def __init__(self,initdata):
        self.data = initdata
        self.next = None

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

    def setData(self,newdata):
        self.data = newdata

    def setNext(self,newnext):
        self.next = newnext
```

*Listing 1*

我们创建一个 Node 对象

```
>>> temp = Node(93)
>>> temp.getData()
93
```

Python 引用值 None 将在 Node 类和链表本身发挥重要作用。引用 None 代表没有下一个节点。请注意在构造函数中，最初创建的节点 next 被设置为 None。有时这被称为 `接地节点`，因此我们使用标准接地符号表示对 None 的引用。将 None 显式的分配给初始下一个引用值是个好主意。

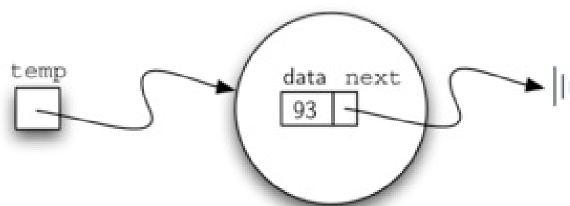



*Figure 3*



*Figure 4*

## 3.21.2.Unordered List 类

如上所述，无序列表将从一组节点构建，每个节点通过显式引用链接到下一个节点。只要我们知道在哪里找到第一个节点（包含第一个项），之后的每个项可以通过连续跟随下一个链接找到。考虑到这一点，UnorderedList 类必须保持对第一个节点的引用。Listing 2 显示了构造函数。注意，每个链表对象将维护对链表头部的单个引用。

```
class UnorderedList:

    def __init__(self):
        self.head = None
```

*Listing 2*

我们构建一个空的链表。赋值语句

```
>>> mylist = UnorderedList()
```

创建如 Figure 5 所示的链表。正如我们在 Node 类中讨论的，特殊引用 None 将再次用于表示链表的头部不引用任何内容。最终，先前给出的示例列表如 Figure 6 所示的链接列表表示。链表的头指代列表的第一项的第一节点。反过来，该节点保存对下一个节点（下一个项）的引用，等等。重要的是注意链表类本身不包含任何节点对象。相反，它只包含对链接结构中第一个节点的单个引用。




*Figure 5*

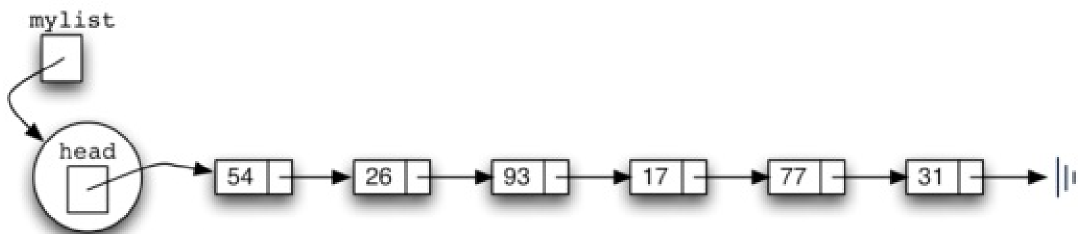



*Figure 6*

Listing 3 中所示的 isEmpty 方法只是检查链表头是否是 None 的引用。 布尔表达式 self.head == None 的结果只有在链表中没有节点时才为眞。由于新链表为空，因此构造函数和空检查必须彼此一致。这显示了使用引用 None 来表示链接结构的 `end` 的优点。在 Python 中，None 可以与任何引用进行比较。如果它们都指向相同的对象，则两个引用是相等的。我们将在其他方法中经常使用它。

```
def isEmpty(self):
    return self.head == None
```

*Listing 3*

那么，我们如何将项加入我们的链表？我们需要实现 add 方法。然而，在我们做这一点之前，我们需要解决在链表中哪个位置放置新项的重要问题。由于该链表是无序的，所以新项相对于已经在列表中的其他项的特定位置并不重要。 新项可以在任何位置。考虑到这一点，将新项放在最简单的位置是有意义的。

回想一下，链表结构只为我们提供了一个入口点，即链表的头部。所有其他节点只能通过访问第一个节点，然后跟随下一个链接到达。这意味着添加新节点的最简单的地方就在链表的头部。 换句话说，我们将新项作为链表的第一项，现有项将需要链接到这个新项后。

Figure 6 展示了链表调用多次 add 函数的操作

```
>>> mylist.add(31)
>>> mylist.add(77)
>>> mylist.add(17)
>>> mylist.add(93)
>>> mylist.add(26)
>>> mylist.add(54)
```

*Figure 6*

因为 31 是添加到链表的第一个项，它最终将是链表中的最后一个节点，因为每个其他项在其前面添加。此外，由于 54 是添加的最后一项，它将成为链表的第一个节点中的数据值。

add 方法如 Listing 4 所示。链表的每项必须驻留在节点对象中。第 2 行创建一个新节点并将该项作为其数据。现在我们必须通过将新节点链接到现有结构中来完成该过程。这需要两个步骤，如 Figure 7 所示。步骤1（行3）更改新节点的下一个引用以引用旧链表的第一个节点。现在，链表的其余部分已经正确地附加到新节点，我们可以修改链表的头以引用新节点。第 4 行中的赋值语句设置列表的头。

上述两个步骤的顺序非常重要。如果第 3 行和第 4 行的顺序颠倒，会发生什么？如果链表头部的修改首先发生，则结果可以在 Figure 8 中看到。由于 head 是链表节点的唯一外部引用，所有原始节点都将丢失并且不能再被访问。

```
def add(self,item):
    temp = Node(item)
    temp.setNext(self.head)
    self.head = temp
```

*Listing 4*

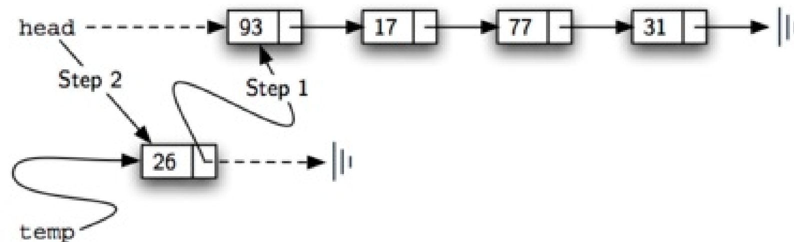



*Figure 7*

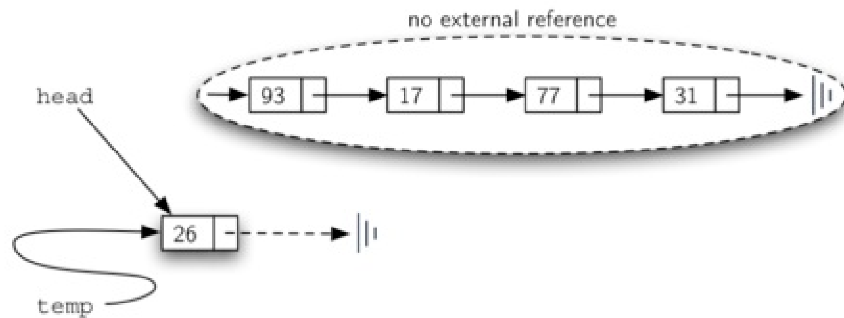



*Figure 8*

我们将实现的下面的方法 - `size` ，`search` 和 `remove` - 都基于一种称为链表遍历的技术。遍历是指系统地访问每个节点的过程。为此，我们使用从链表中第一个节点开始的外部引用。当我们访问每个节点时，我们通过“遍历”下一个引用来移动到对下一个节点的引用。

要实现 `size` 方法，我们需要遍历链表并对节点数计数。Listing 5 展示了用于计算列表中节点数的 Python 代码。外部引用称为 `current` ，并在第二行被初始化到链表的头部。开始的时候，我们没有看到任何节点，所以计数设置为 0 。第 4-6 行实际上实现了遍历。只要当前引用没到链表的结束位置（None），我们通过第 6 行中的赋值语句将当前元素移动到下一个节点。再次，将引用与 None 进行比较的能力是非常有用的。每当 `current` 移动到一个新的节点，我们加 1 以计数。最后， `count` 在迭代停止后返回。Figure 9 展示了处理这个链表的过程。

```
def size(self):
    current = self.head
    count = 0
    while current != None:
        count = count + 1
        current = current.getNext()

    return count
```

*Listing 5*

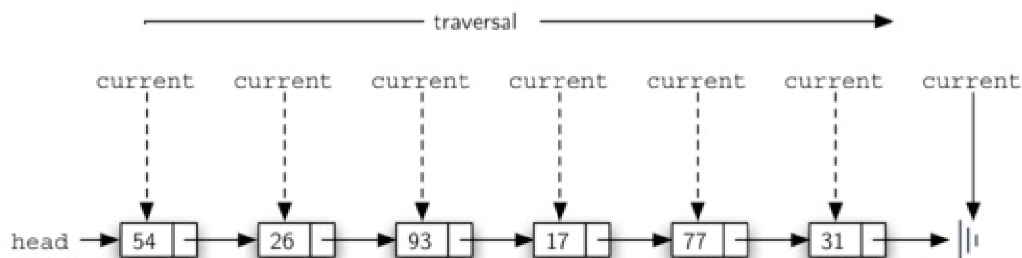



*Figure 9*

在链表中搜索也使用遍历技术。当我们访问链表中的每个节点时，我们将询问存储在其中的数据是否与我们正在寻找的项匹配。然而，在这种情况下，我们不必一直遍历到列表的末尾。事实上，如果我们到达链表的末尾，这意味着我们正在寻找的项不存在。此外，如果我们找到项，没有必要继续。

Listing 6 展示了搜索方法的实现。和在 `size` 方法中一样，遍历从列表的头部开始初始化（行2）。我们还使用一个布尔变量叫 `found`，标记我们是否找到了正在寻找的项。因为我们还没有在遍历开始时找到该项，`found` 设置为 False（第3行）。第4行中的迭代考虑了上述两个条件。只要有更多的节点访问，而且我们没有找到正在寻找的项，我们就继续检查下一个节点。第 5 行检查数据项是否存在于当前节点中。如果存在，`found` 设置为 True。

```
def search(self,item):
    current = self.head
    found = False
    while current != None and not found:
        if current.getData() == item:
            found = True
        else:
            current = current.getNext()

    return found
```

*Listing 6*

作为一个例子，试试调用 `search` 方法来查找 item 17

```
>>> mylist.search(17)
True
```

因为 17 在列表中，所以遍历过程需要移动到包含 17 的节点。此时，`found` 变量设置为 True，while 条件将失败，返回值。这个过程可以在 Figure 10中看到。

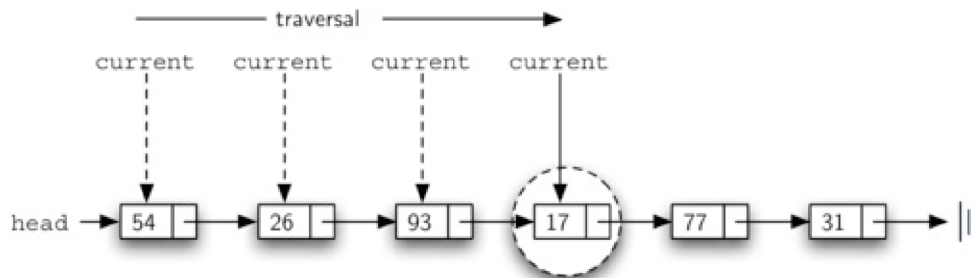



*Figure 10*

`remove` 方法需要两个逻辑步骤。首先，我们需要遍历列表寻找我们要删除的项。一旦我们找到该项（我们假设它存在），删除它。第一步非常类似于搜索。从设置到链表头部的外部引用开始，我们遍历链接，直到我们发现正在寻找的项。因为我们假设项存在，我们知道迭代将在 `current` 变为 `None` 之前停止。这意味着我们可以简单地使用 `found` 布尔值。

当 `found` 变为 True 时，`current` 将是对包含要删除的项的节点的引用。但是我们如何删除呢？一种方法是用标示该项目不再存在的某个标记来替换项目的值。这种方法的问题是节点数量将不再匹配项数量。最好通过删除整个节点来删除该项。

为了删除包含项的节点，我们需要修改上一个节点中的链接，以便它指向当前之后的节点。不幸的是，链表遍历没法回退。因为 `current` 指我们想要进行改变的节点之前的节点，所以进行修改太迟了。

这个困境的解决方案是在我们遍历链表时使用两个外部引用。`current` 将像之前一样工作，标记遍历的当前位置。新的引用，我们叫 `previous`，将总是传递 `current` 后面的一个节点。这样，当 `current` 停止在要被去除的节点时，`previous` 将引用链表中用于修改的位置。

Listing 7 展示了完整的 `remove` 方法。第 2-3 行给这两个引用赋初始值。注意，`current` 在链表头处开始，和在其他遍历示例中一样。然而，`previous` 假定总是在 `current` 之后一个节点。因此，由于在 `previous` 之前没有节点，所以之前的值将为 None（见 Figure 11）。`found` 的布尔变量将再次用于控制迭代。

在第 6-7 行中，我们检查存储在当前节点中的项是否是我们希望删除的项。如果是，`found` 设置为 True 。如果我们没有找到该项，则 `previous` 和 `current` 都必须向前移动一个节点。同样，这两个语句的顺序是至关重要的。`previous` 必须先将一个节点移动到 `current`

的位置。此时，才可以移动 `current` 。这个过程通常被称为“英寸蠕动”，因为 `previous` 必须赶上 `current` ，然后 `current` 前进。Figure 12 展示了 `previous` 和 `current` 的移动，它们沿着链表向下移动，寻找包含值 17 的节点。

```
def remove(self,item):
    current = self.head
    previous = None
    found = False
    while not found:
        if current.getData() == item:
            found = True
        else:
            previous = current
            current = current.getNext()

    if previous == None:
        self.head = current.getNext()
    else:
        previous.setNext(current.getNext())
```

*Listing 7*

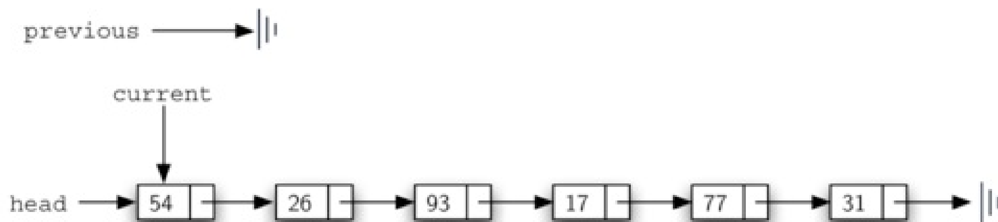



*Figure 11*

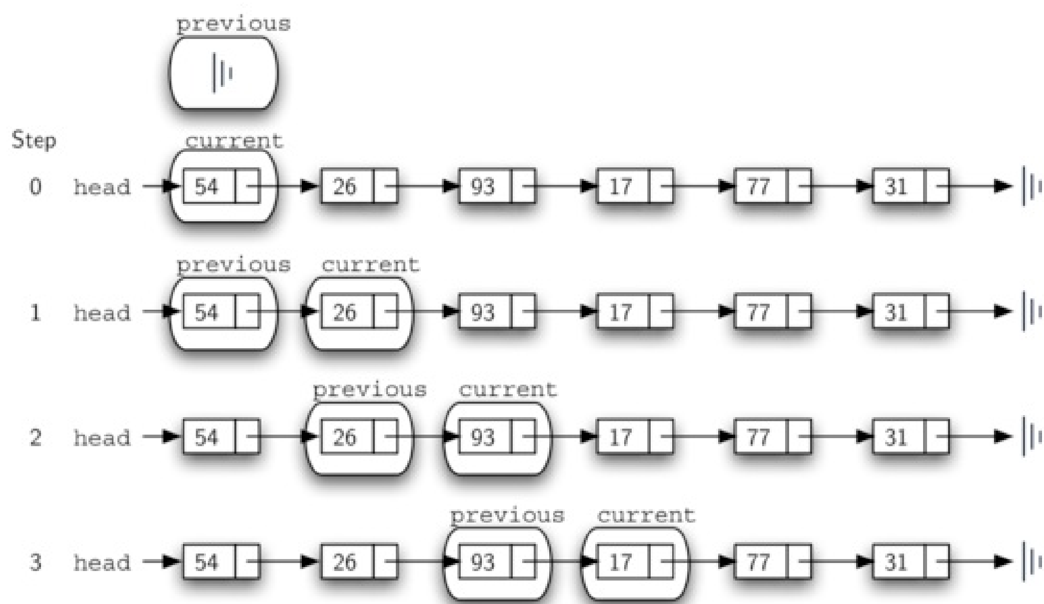



*Figure 12*

一旦 `remove` 的搜索步骤已经完成，我们需要从链表中删除该节点。 Figure 13 展示了要修改的链接。但是，有一个特殊情况需要解决。 如果要删除的项目恰好是链表中的第一个项，则 `current` 将引用链接列表中的第一个节点。这也意味着 `previous` 是 None。 我们先前说过， `previous` 是一个节点，它的下一个节点需要修改。在这种情况下，不是 `previous` ，而是链表的 `head` 需要改变（见 Figure 14）。

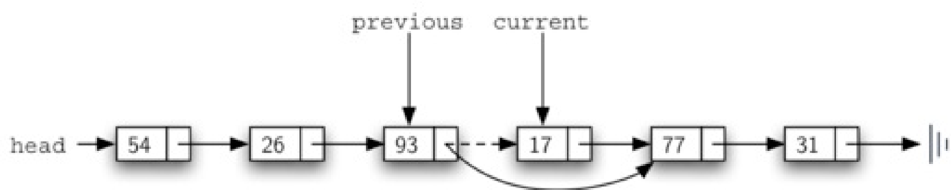



*Figure 13*

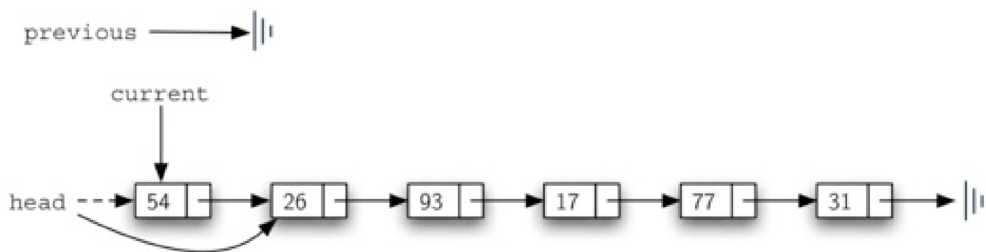



*Figure 14*

第 12 行检查是否处理上述的特殊情况。如果 `previous` 没有移动，当 `found` 的布尔变为 True 时，它仍是 None。 在这种情况下（行13），链表的 `head` 被修改以指代当前节点之后的节点，实际上是从链表中移除第一节点。 但是，如果 `previous` 不为 None，则要删除的

节点位于链表结构的下方。 在这种情况下， `previous` 的引用为我们提供了下一个引用更改的节点。第 15 行使用之前的 `setNext` 方法完成删除。注意，在这两种情况下，引用更改的目标是 `current.getNext()` 。 经常出现的一个问题是，这里给出的两种情况是否也将处理要移除的项在链表的最后节点中的情况。我们留给你思考。

# 3.22.有序列表抽象数据结构

我们现在将考虑一种称为有序列表的列表类型。例如，如果上面所示的整数列表是有序列表（升序），则它可以写为 `17,26,31,54,77和93` 。由于 17 是最小项，它占据第一位置。同样，由于 93 是最大的，它占据最后的位置。

有序列表的结构是项的集合，其中每个项保存基于项的一些潜在特性的相对位置。排序通常是升序或降序，并且我们假设列表项具有已经定义的有意义的比较运算。许多有序列表操作与无序列表的操作相同。

- OrderedList() 创建一个新的空列表。它不需要参数，并返回一个空列表。
- add(item) 向列表中添加一个新项。它需要 item 作为参数，并不返回任何内容。假定该 item 不在列表中。
- remove(item) 从列表中删除该项。它需要 item 作为参数并修改列表。假设项存在于列表中。
- search(item) 搜索列表中的项目。它需要 item 作为参数，并返回一个布尔值。
- isEmpty() 检查列表是否为空。它不需要参数，并返回布尔值。
- size（）返回列表中的项数。它不需要参数，并返回一个整数。
- index(item) 返回项在列表中的位置。它需要 item 作为参数并返回索引。假定该项在列表中。
- pop() 删除并返回列表中的最后一个项。假设该列表至少有一个项。
- pop(pos) 删除并返回位置 pos 处的项。它需要 pos 作为参数并返回项。假定该项在列表中。

# 3.23.实现有序列表

为了实现有序列表，我们必须记住项的相对位置是基于一些潜在的特性。上面给出的整数的有序列表 `17,26,31,54,77` 和 `93` 可以由 Figure 15 所示的链接结构表示。节点和链接结构表示项的相对位置。

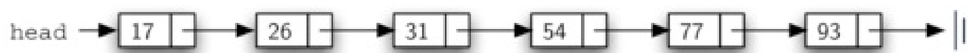



*Figure 15*

为了实现 OrderedList 类，我们将使用与前面看到的无序列表相同的技术。再次，`head` 的引用为 `None` 表示为空链表（参见 Listing 8）。

```
class OrderedList:
    def __init__(self):
        self.head = None
```

*Listing 8*

当我们考虑有序列表的操作时，我们应该注意，`isEmpty` 和 `size` 方法可以与无序列表一样实现，因为它们只处理链表中的节点数量，而不考虑实际项值。同样，`remove` 方法将正常工作，因为我们仍然需要找到该项，然后删除它。剩下的两个方法，`search` 和 `add`，将需要一些修改。

搜索无序列表需要我们一次遍历一个节点，直到找到我们正在寻找的节点或者没找到节点（None）。事实证明，相同的方法在有序列表也有效。然而，在项不在链表中的情况下，我们可以利用该顺序来尽快停止搜索。

例如，Figure 16 展示了有序链表搜索值 45 。从链表的头部开始遍历，首先与 `17` 进行比较。由于 `17` 不是我们正在寻找的项，移动到下一个节点 `26` 。再次，这不是我们想要的，继续到 `31`，然后再到 `54` 。在这一点上，有一些不同。由于 `54` 不是我们正在寻找的项，我们以前的方法是继续向前迭代。然而，由于这是有序列表，一旦节点中的值变得大于我们正在搜索的项，搜索就可以停止并返回 False 。该项不可能存在于后面的链表中。

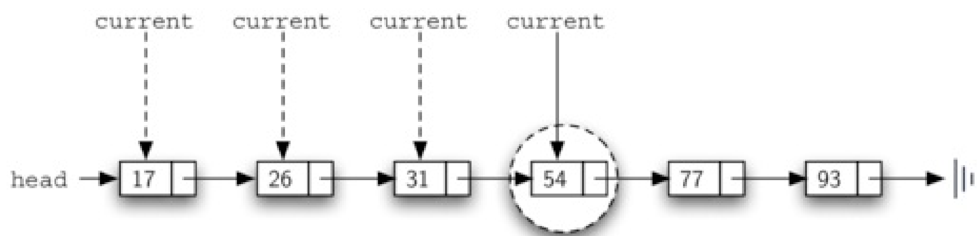


*Figure 16*

Listing 9 展示了完整的搜索方法。通过添加另一个布尔变量 `stop` 并将其初始化为 `False` （第4行），很容易合并上述新条件。 当 `stop` 是 `False` （不停止）时，我们可以继续在列表中前进（第5行）。如果发现任何节点包含大于我们正在寻找的项的数据，我们将 `stop` 设置为 `True` （第9-10行）。其余行与无序列表搜索相同。

```
def search(self,item):
    current = self.head
    found = False
    stop = False
    while current != None and not found and not stop:
        if current.getData() == item:
            found = True
        else:
            if current.getData() > item:
                stop = True
            else:
                current = current.getNext()

    return found
```

*Listing 9*

最重要的需要修改的方法是 `add` 。 回想一下，对于无序列表， `add` 方法可以简单地将新节点放置在链表的头部。 这是最简单的访问点。 不幸的是，这将不再适用于有序列表。需要在现有的有序列表中查找新项所属的特定位置。

假设我们有由 `17,26,54,77` 和 `93` 组成的有序列表，并且我们要添加值 `31` 。 `add` 方法必须确定新项属于 `26` 到 `54` 之间。Figure 17 展示了我们需要的设置。正如我们前面解释的，我们需要遍历链表，寻找添加新节点的地方。我们知道，当我们迭代完节点（ `current` 变为 None）或 `current` 节点的值变得大于我们希望添加的项时，我们就找到了该位置。在我们的例子中，看到值 `54` 我们停止迭代。

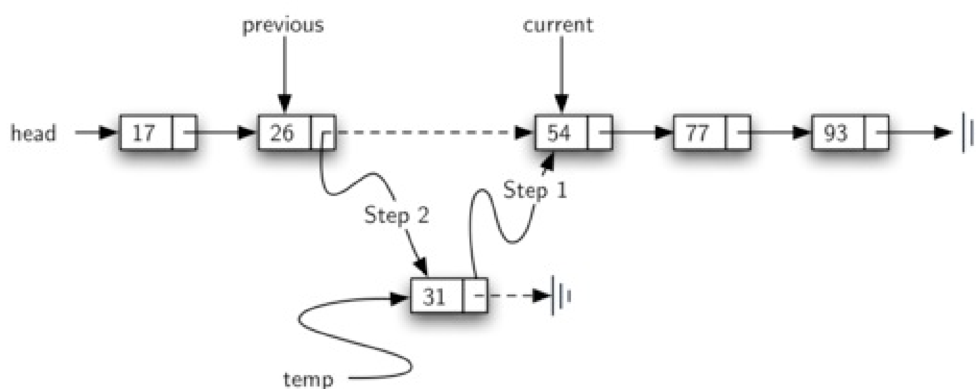



*Figure 17*

正如我们看到的无序列表，有必要有一个额外的引用，再次称为 `previous` ，因为 `current` 不会提供对修改的节点的访问。 Listing 10 展示了完整的 `add` 方法。 行 2-3 设置两个外部引用，行 9-10 允许 `previous` 每次通过迭代跟随 `current` 节点后面 。 条件（行5）允许迭代继续，只要有更多的节点，并且当前节点中的值不大于该项。 在任一种情况下，当迭代失败时，我们找到了新节点的位置。

该方法的其余部分完成 Figure17 所示的两步过程。一旦为该项创建了新节点，剩下的唯一问题是新节点是否将被添加在链表的开始处或某个中间位置。再次， `previous == None` （第13行）可以用来提供答案。

```
def add(self,item):
    current = self.head
    previous = None
    stop = False
    while current != None and not stop:
        if current.getData() > item:
            stop = True
        else:
            previous = current
            current = current.getNext()

    temp = Node(item)
    if previous == None:
        temp.setNext(self.head)
        self.head = temp
    else:
        temp.setNext(current)
        previous.setNext(temp)
```

*Listing 10*

## 3.23.1.链表分析

为了分析链表操作的复杂性，我们需要考虑它们是否需要遍历。考虑具有 n 个节点的链表。 `isEmpty` 方法是 O(1)，因为它需要一个步骤来检查头的引用为 `None` 。另一方面， `size` 将总是需要 n 个步骤，因为不从头到尾地移动没法知道有多少节点在链表中。因此，长度为 O(n)。将项添加到无序列表始终是O(1)，因为我们只是将新节点放置在链表的头部。但是，搜索和删除，以及添加有序列表，都需要遍历过程。虽然平均他们可能只需要遍历节点的一半，这些方法都是 O(n)，因为在最坏的情况下，都将处理列表中的每个节点。

你可能还注意到此实现的性能与早前针对 Python 列表给出的实际性能不同。这表明链表不是 Python 列表的实现方式。 Python 列表的实际实现基于数组的概念。我们在第 8 章中更详细地讨论这个问题。

# 3.24.总结

- 线性数据结构以有序的方式保存它们的数据。
- 栈是维持 LIFO，后进先出，排序的简单数据结构。
- 栈的基本操作是 `push` ， `pop` 和 `isEmpty` 。
- 队列是维护 FIFO（先进先出）排序的简单数据结构。
- 队列的基本操作是 `enqueue` ， `dequeue` 和 `isEmpty` 。
- 前缀，中缀和后缀都是写表达式的方法。
- 栈对于设计计算解析表达式算法非常有用。
- 栈可以提供反转特性。
- 队列可以帮助构建定时仿眞。
- 模拟使用随机数生成器来创建眞实情况，并帮助我们回答“假设”类型的问题。
- `Deques` 是允许类似栈和队列的混合行为的数据结构。
- `deque` 的基本操作是 `addFront` ， `addRear` ， `removeFront` ， `removeRear` 和 `isEmpty` 。
- 列表是项的集合，其中每个项目保存相对位置。
- 链表实现保持逻辑顺序，而不需要物理存储要求。
- 修改链表头是一种特殊情况。

# 4.1. 目标

本章的目标如下：

- 要理解可能难以解决的复杂问题有一个简单的递归解决方案。
- 学习如何递归地写出程序。
- 理解和应用递归的三个定律。
- 将递归理解为一种迭代形式。
- 实现问题的递归公式化。
- 了解计算机系统如何实现递归。

# 4.2.什么是递归

递归是一种解决问题的方法，将问题分解为更小的子问题，直到得到一个足够小的问题可以被很简单的解决。通常递归涉及函数调用自身。递归允许我们编写优雅的解决方案，解决可能很难编程的问题。

# 4.3.计算整数列表和

我们将以一个简单的问题开始，你已经知道如何不使用递归解决。 假设你想计算整数列表的总和，例如：`[1,3,5,7,9]` 。 计算总和的迭代函数见ActiveCode 1。函数使用累加器变量（`theSum`）来计算列表中所有整数的和，从 0 开始，加上列表中的每个数字。

```
def listsum(numList):
    theSum = 0
    for i in numList:
        theSum = theSum + i
    return theSum

print(listsum([1,3,5,7,9]))
```

*Activecode 1*

假设没有 `while` 循环或 `for` 循环。你将如何计算整数列表的总和？如果你是一个数学家，你可能开始回忆加法是一个函数，这个函数定义了两个整数类型的参数。故将列表和问题从加一个列表重新定义为加一对整数，我们可以把列表重写为一个完全括号表达式。如下所示：

$$((((1+3)+5)+7)+9)$$

我们也可以把表达式用另一种方式括起来

$$(1+(3+(5+(7+9))))$$

注意，最内层的括号（7 + 9）我们可以没有循环或任何特殊的结构来解决它。 事实上，我们可以使用以下的简化序列来计算最终的和。

$$\begin{aligned} total = & \ (1+(3+(5+(7+9)))) \\ total = & \ (1+(3+(5+16))) \\ total = & \ (1+(3+21)) \\ total = & \ (1+24) \\ total = & \ 25 \end{aligned}$$

我们如何能把这个想法变成一个 Python 程序？ 首先，让我们以 Python 列表的形式重述求和问题。 我们可以说列表 `numList` 的和是列表的第一个元素 `numList[0]` 和列表其余部分 `numList [1:]` 之和的总和。 以函数形式表述：

$$listSum(numList) = first(numList) + listSum(rest(numList))$$

在这个方程式中，`first(numList)` 返回列表的第一个元素，`rest(numList)` 返回除第一个元素之外的所有元素列表。这很容易在 Python 中表示，如 ActiveCode 2 中所示。

```
def listsum(numList):
   if len(numList) == 1:
        return numList[0]
   else:
        return numList[0] + listsum(numList[1:])

print(listsum([1,3,5,7,9]))
```

*Active code 2*

在这个清单中有几个关键地方。 首先，在第 2 行，我们检查列表是否为一个元素。这个检查是至关重要的，是我们的函数的转折子句。 长度为 1 的列表和是微不足道的; 它只是列表中的数字。 第二，在第 5 行函数调用自己！ 这就是我们称 listum 算法递归的原因。递归函数是调用自身的函数。

Figure 1 展示了对列表 `[1,3,5,7,9]` 求和所需的一系列递归调用。 你应该把这一系列的调用想象成一系列的简化。 每次我们进行递归调用时，我们都会解决一个较小的问题，直到达到问题不能减小的程度。

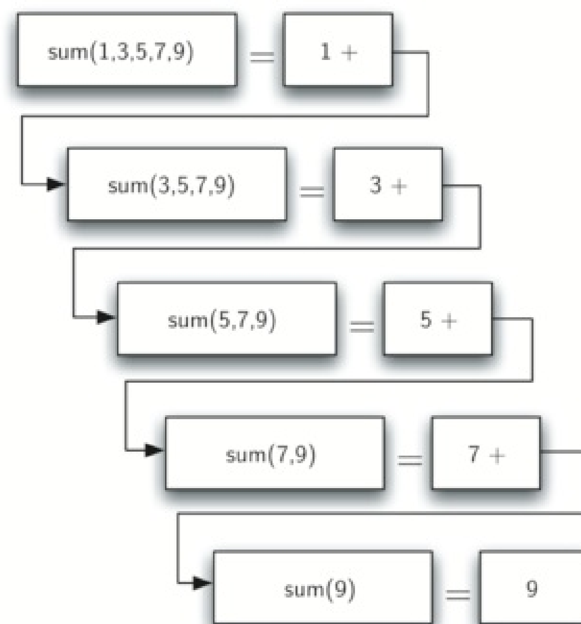



*Figure 1*

当我们到达简单问题的点，我们开始拼凑每个小问题的答案，直到初始问题解决。Figure 2 展示了在 `listsum` 通过一系列调用返回的过程中执行的 add 操作。当 `listsum` 从最顶层返回时，我们就有了整个问题的答案。

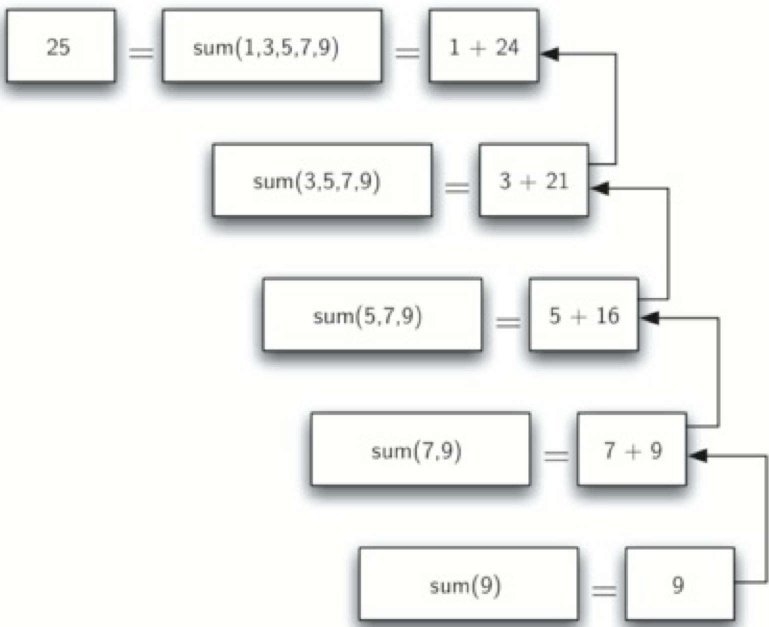



*Figure 2*

# 4.4.递归的三定律

像阿西莫夫机器人，所有递归算法必须服从三个重要的定律：

1. 递归算法必须具有基本情况。
2. 递归算法必须改变其状态并向基本情况靠近。
3. 递归算法必须以递归方式调用自身。

让我们更详细地看看每一个定律，看看它如何在 `listsum` 算法中使用。首先，基本情况是算法停止递归的条件。基本情况通常是足够小以直接求解的问题。在 `listsum` 算法中，基本情况是长度为 1 的列表。

为了遵守第二定律，我们必须将算法向基本情况的状态改变。状态的改变意味着该算法正在使用的一些数据被修改。通常，表示问题的数据在某种程度上变小。在 `listsum` 算法中，我们的主要数据结构是一个列表，因此我们必须将我们的状态转换工作集中在列表上。因为基本情况是长度 1 的列表，所以朝向基本情况的自然进展是缩短列表。在 Activecode 2 第五行，我们调用 `listsum` 生成一个较短的列表。

最后的法则是算法必须调用自身。这是递归的定义。递归对于许多新手程序员来说是一个混乱的概念。作为一个新手程序员，你已经知道函数是有益的，因为你可以将一个大问题分解成较小的问题。较小的问题可以通过编写一个函数来解决。我们用一个函数解决问题，但该函数通过调用自己解决问题！该逻辑不是循环;递归的逻辑是通过将问题分解成更小和更容易的问题来解决的优雅表达。

在本章的剩余部分，我们将讨论更多递归的例子。在每种情况下，我们将集中于使用递归的三个定律来设计问题的解决方案。

# 4.5.整数转换为任意进制字符串

假设你想将一个整数转换为一个二进制和十六进制字符串。例如，将整数 `10` 转换为十进制字符串表示为 `10` ，或将其字符串表示为二进制 `1010` 。虽然有很多算法来解决这个问题，包括在栈部分讨论的算法，但递归的解决方法非常优雅。

让我们看一个十进制数 `769` 的具体示例。假设我们有一个对应于前 `10` 位数的字符序列，例如 `convString =“0123456789”` 。 通过在序列中查找，很容易将小于 `10` 的数字转换为其等效的字符串。例如，如果数字为 `9` ，则字符串为 convString[9] 或 “9”。如果我们将数字 `769` 分成三个单个位数字， `7` ， `6` 和 `9` ，那么将其转换为字符串很简单。数字小于 10 听起来像一个好的基本情况。

知道我们的基本情况是什么意味着整个算法将分成三个部分：

1. 将原始数字减少为一系列单个位数字。
2. 使用查找将单个位数字数字转换为字符串。
3. 将单个位字符串连接在一起以形成最终结果。

下一步是找到改变其状态的方法并向基本情况靠近。由于我们示例为整数，所以考虑什么数学运算可以减少一个数字。最可能的候选是除法和减法。虽然减法可能可以实现，但我们不清楚应该减去多少。使用余数的整数除法为我们提供了一个明确的方向。让我们看看如果我们将一个数字除以我们试图转换的基数，会发生什么。

使用整数除法将 `769` 除以 `10` ，我们得到 `76` ，余数为 `9` 。这给了我们两个好的结果。首先，余数是小于我们的基数的数字，可以通过查找立即转换为字符串。第二，我们得到的商小于原始数字，并让我们靠近具有小于基数的单个数字的基本情况。现在我们的工作是将 `76` 转换为其字符串表示。再次，我们使用商和余数分别获得 `7` 和 `6` 的结果。最后，我们将问题减少到转换 `7` ，我们可以很容易地做到，因为它满足 `n < base` 的基本条件，其中 `base = 10` 。我们刚刚执行的一系列操作如 Figure 3 所示。请注意，余数位于图右侧框中。

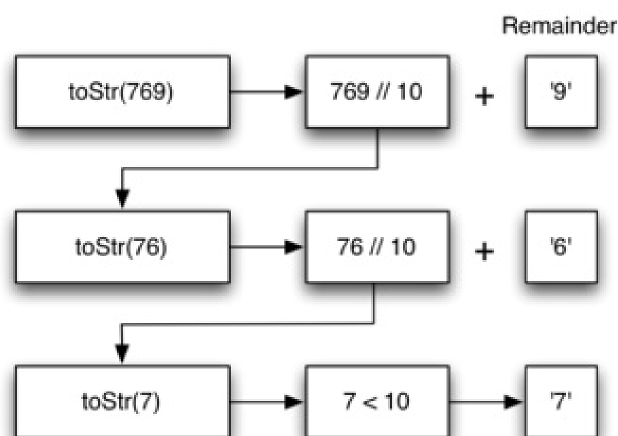


*Figure 3*

ActiveCode 1 展示了实现上述算法的 Python 代码， 以 2 到 16 之间的任何基数为参数。

```
def toStr(n,base):
   convertString = "0123456789ABCDEF"
   if n < base:
      return convertString[n]
   else:
      return toStr(n//base,base) + convertString[n%base]

print(toStr(1453,16))
```

请注意，在第 3 行中，我们检查基本情况，其中 n 小于我们要转换的基数。 当我们检测到基本情况时，我们停止递归，并简单地从 convertString 序列返回字符串。 在第 6 行中，我们满足第二和第三定律 - 递归调用和减少除法问题大小。

让我们再次跟踪算法; 这次我们将数字 10 转换为其基数为 2 的字符串（“1010”）。

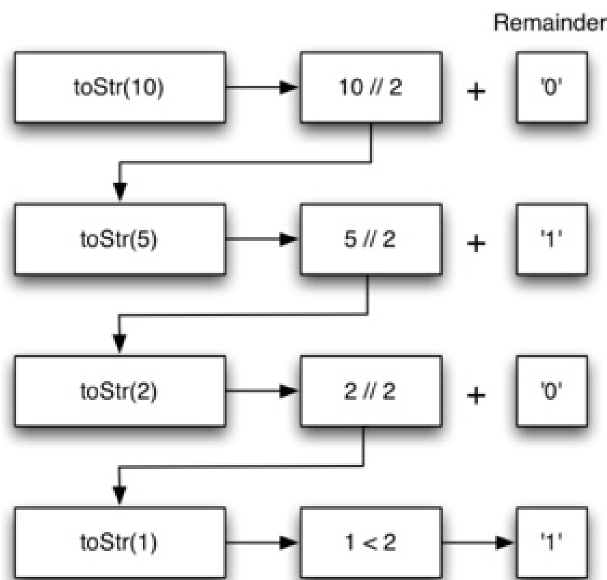



*Figure 4*

Figure 4 显示我们得到的结果，但看起来数字是错误的顺序。该算法是正确的，因为我们首先在第 6 行进行递归调用，然后我们添加余数的字符串形式。 如果我们反向返回 convertString 查找并返回 toStr 调用，则生成的字符串将是反向的！通过延后连接操作直到递归调用返回，我们可以得到正确顺序的结果。这应该能使你想起你在上一章中讨论的栈。

# 4.6.栈帧：实现递归

假设不是将递归调用的结果与来自 convertString 的字符串拼接到 toStr，我们修改了算法，以便在进行递归调用之前将字符串入栈。此修改的算法的代码展示在 ActiveCode 1 中。

```
from pythonds.basic.stack import Stack

rStack = Stack()

def toStr(n,base):
    convertString = "0123456789ABCDEF"
    while n > 0:
        if n < base:
            rStack.push(convertString[n])
        else:
            rStack.push(convertString[n % base])
        n = n // base
    res = ""
    while not rStack.isEmpty():
        res = res + str(rStack.pop())
    return res

print(toStr(1453,16))
```

*ActiveCode 1*

每次我们调用 toStr，我们在栈上推入一个字符。回到前面的例子，我们可以看到在第四次调用 toStr 之后，栈看起来像 Figure 5 。注意，现在我们可以简单地将字符从栈中弹出，并将它们连接成最终结果 “1010”。

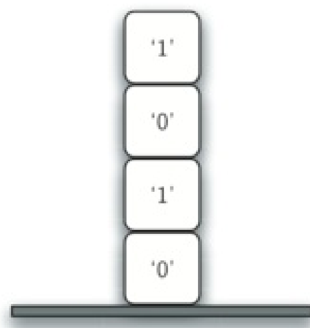



*Figure 5*

前面的例子让我们了解了 Python 如何实现一个递归函数调用。 当在 Python 中调用函数时，会分配一个栈来处理函数的局部变量。当函数返回时，返回值留在栈的顶部，以供调用函数访问。 Figure 6 说明了第 4 行返回语句后的调用栈。

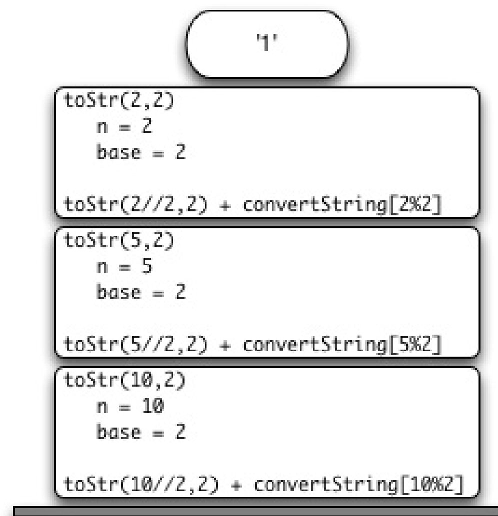



*Figure 6*

注意，对 `toStr(2//2,2)` 的调用在栈上返回值为 “1”。 然后，在表达式 `“1” + convertString[2%2]` 中使用此返回值替换函数调用 `(toStr(1,2))` ，这将在栈顶部留下字符串 “10”。 这样，Python 调用栈就代替了我们在 Listing 4 中明确使用的栈。在我们的列表求和示例中，你可以认为栈上的返回值取代了累加器变量。

栈帧还为函数使用的变量提供了一个作用域。 即使我们重复地调用相同的函数，每次调用都会为函数本地的变量创建一个新的作用域。

# 4.7.介绍：可视化递归

在上一节中，我们讨论了一些使用递归很容易解决的问题; 然而，我们可能很难找到一个模型或一种可视化方法知道在递归函数中发生了什么。这使得递归难以让人掌握。在本节中，我们将看到几个使用递归绘制一些有趣图片的例子。当你看到这些图片的形状，你会对递归过程有新的认识，可能有助于巩固你对递归理解。

我们使用的插图的工具是 Python 的 `turtle` 模块称为 `turtle` 。`turtle` 是 Python 所有版本的标准库，并且非常易于使用。比喻很简单。你可以创建一只乌龟，乌龟能前进，后退，左转，右转等。乌龟可以让它的尾巴或上或下。当乌龟的尾巴向下，它移动时会画一条线。为了增加乌龟的艺术价值，你可以改变尾巴的宽度以及尾巴浸入的墨水的颜色。

这里有一个简单的例子来说明龟图形基础。我们将使用 `turtle` 模块递归绘制螺旋。 见 ActiveCode 1。导入 `turtle` 模块后，我们创建一个乌龟。当乌龟被创建时，它也创建一个窗口来绘制。接下来我们定义 `drawSpiral` 函数。这个简单函数的基本情况是当我们想要绘制的线的长度（由 len 参数给出）减小到零或更小时。如果线的长度大于零，我们让乌龟以 len 单位前进，然后向右转 90 度。当我们再次调用 `drawSpiral` 并缩短长度时递归。在 ActiveCode 1 结束时，你会注意到我们调用函数 `myWin.exitonclick()` ，这是一个方便的缩小窗口的方法，使乌龟进入等待模式，直到你单击窗口，然后程序清理并退出。

```
import turtle

myTurtle = turtle.Turtle()
myWin = turtle.Screen()

def drawSpiral(myTurtle, lineLen):
    if lineLen > 0:
        myTurtle.forward(lineLen)
        myTurtle.right(90)
        drawSpiral(myTurtle,lineLen-5)

drawSpiral(myTurtle,100)
myWin.exitonclick()
```

这是关于你知道的所有龟图形，以制作一些令人印象深刻的涂鸦。我们的下一个程序，将绘制一个分形树。分形来自数学的一个分支，并且与递归有很多共同之处。分形的定义是，当你看着它时，无论你放大多少，分形有相同的基本形状。大自然的一些例子是大陆的海岸线，雪花，山脉，甚至树木或灌木。这些自然现象中的许多的分形性质使得程序员能够为计算机生成的电影生成非常逼眞的风景。在我们的下一个例子中，将生成一个分形树。

要理解这如何工作，需要想一想如何使用分形词汇来描述树。记住，我们上面说过，分形是在所有不同的放大倍率下看起来是一样的。如果我们将它翻译成树木和灌木，我们可能会说，即使一个小树枝也有一个整体树的相同的形状和特征。有了这个想法，我们可以说一棵

树是树干，一棵较小的树向右走，另一棵较小的树向左走。如果你用递归的思想考虑这个定义，这意味着我们将树的递归定义应用到较小的左树和右树。

让我们把这个想法转换成一些 Python 代码。Listing 1 展示了如何使用我们的乌龟来生成分形树。让我们更仔细地看一下代码。你会看到在第 5 行和第 7 行，我们正在进行递归调用。在第 5 行，我们在乌龟向右转 20 度之后立即进行递归调用;这是上面提到的右树。然后在第 7 行，乌龟进行另一个递归调用，但这一次后左转 40 度。乌龟必须向左转 40 度的原因是，它需要撤消原来的向右转 20 度，然后再向左转 20 度，以绘制左树。还要注意，每次我们对树进行递归调用时，我们从 `branchLen` 参数中减去一些量; 这是为了确保递归树越来越小。你还应该看到到第 2 行的初始 if 语句是检查 `branchLen` 的基本情况大小。

```
def tree(branchLen,t):
    if branchLen > 5:
        t.forward(branchLen)
        t.right(20)
        tree(branchLen-15,t)
        t.left(40)
        tree(branchLen-10,t)
        t.right(20)
        t.backward(branchLen)
```

*Listing 1*

此树示例的完整程序在 ActiveCode 2 中。在运行代码之前，请思考你希望看到的树形状。看着递归调用，并想想这棵树将如何展开。它会对称地绘制树的右半边和左半边吗？ 它会先绘制右侧然后左侧？

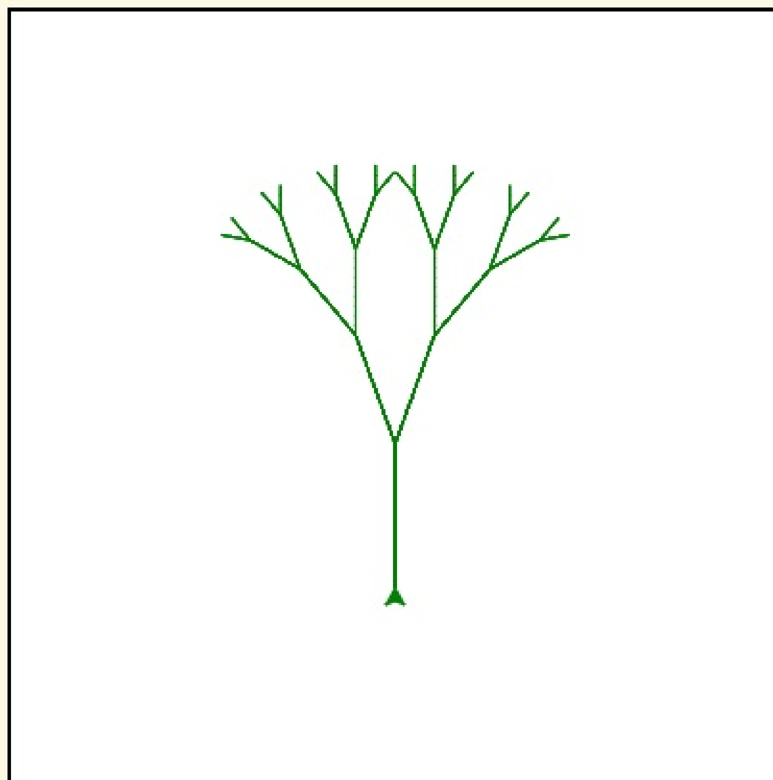
```
import turtle

def tree(branchLen,t):
    if branchLen > 5:
        t.forward(branchLen)
        t.right(20)
        tree(branchLen-15,t)
        t.left(40)
        tree(branchLen-15,t)
        t.right(20)
        t.backward(branchLen)

def main():
    t = turtle.Turtle()
    myWin = turtle.Screen()
    t.left(90)
    t.up()
    t.backward(100)
    t.down()
    t.color("green")
    tree(75,t)
    myWin.exitonclick()

main()
```

*Activecode 2*



注意树上的每个分支点如何对应于递归调用，并注意树的右半部分如何一直绘制到它的最短的树枝。你可以在 Figure 1 中看到这一点。现在，注意程序如何工作，它的方式是直到树的整个右侧绘制完成回到树干。你可以在 Figure 2 中看到树的右半部分。然后绘制树的左侧，但不是尽可能远地向左移动。相反，直到我们进入到左树最小的枝干，左树的右半部分才开始绘制。

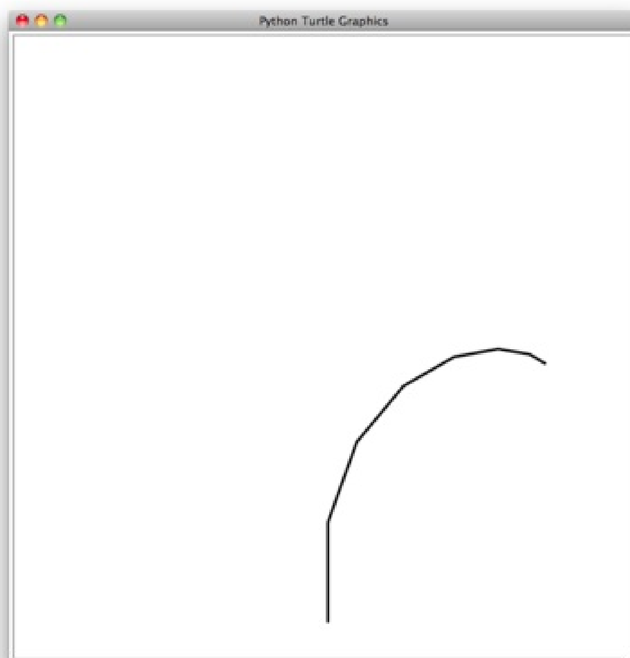



*Figure 1*

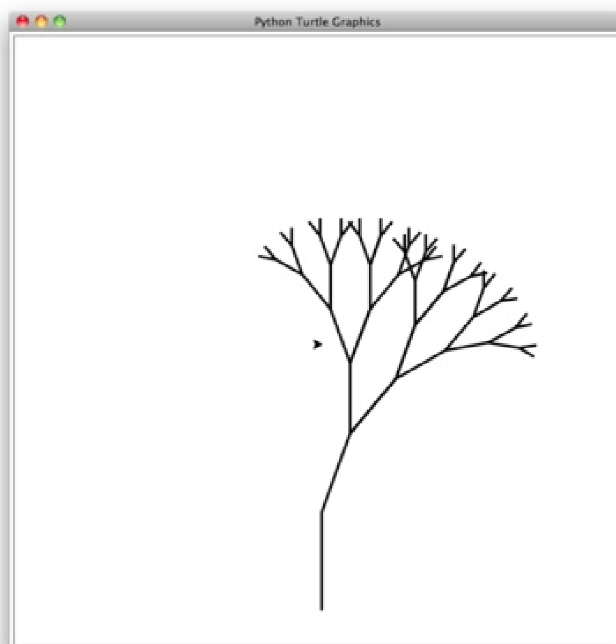



*Figure 2*

这个简单的树程序只是一个起点，你会注意到树看起来不是特别现实，因为自然不像计算机程序那样对称。

# 4.8.谢尔宾斯基三角形

另一个展现自相似性的分形是谢尔宾斯基三角形。 Figure 3 是一个示例。谢尔宾斯基三角形阐明了三路递归算法。用手绘制谢尔宾斯基三角形的过程很简单。 从一个大三角形开始。通过连接每一边的中点，将这个大三角形分成四个新的三角形。忽略刚刚创建的中间三角形，对三个小三角形中的每一个应用相同的过程。 每次创建一组新的三角形时，都会将此过程递归应用于三个较小的角三角形。 如果你有足够的铅笔，你可以无限重复这个过程。在继续阅读之前，你可以尝试运用所描述的方法自己绘制谢尔宾斯基三角形。

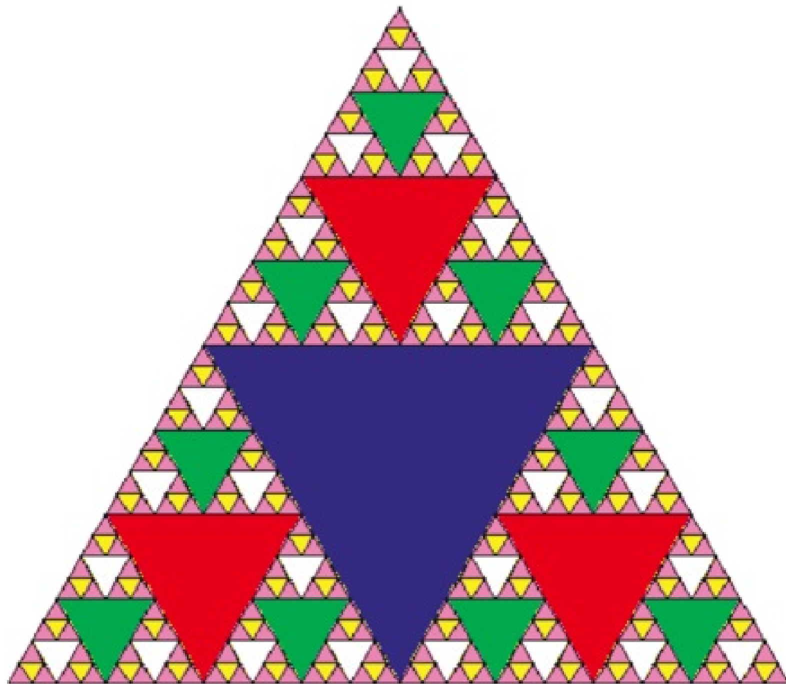


Figure 3: The Sierpinski Triangle

*Figure 3*

因为我们可以无限地应用算法，什么是基本情况？ 我们将看到，基本情况被任意设置为我们想要将三角形划分成块的次数。有时我们把这个数字称为分形的“度”。 每次我们进行递归调用时，我们从度中减去 1，直到 0。当我们达到 0 度时，我们停止递归。在 Figure 3 中生成谢尔宾斯基三角形的代码见 ActiveCode 1。

```
import turtle

def drawTriangle(points,color,myTurtle):
    myTurtle.fillcolor(color)
    myTurtle.up()
    myTurtle.goto(points[0][0],points[0][1])
    myTurtle.down()
    myTurtle.begin_fill()
    myTurtle.goto(points[1][0],points[1][1])
    myTurtle.goto(points[2][0],points[2][1])
    myTurtle.goto(points[0][0],points[0][1])
    myTurtle.end_fill()

def getMid(p1,p2):
    return ( (p1[0]+p2[0]) / 2, (p1[1] + p2[1]) / 2)

def sierpinski(points,degree,myTurtle):
    colormap = ['blue','red','green','white','yellow',
                'violet','orange']
    drawTriangle(points,colormap[degree],myTurtle)
    if degree > 0:
        sierpinski([points[0],
                        getMid(points[0], points[1]),
                        getMid(points[0], points[2])],
                   degree-1, myTurtle)
        sierpinski([points[1],
                        getMid(points[0], points[1]),
                        getMid(points[1], points[2])],
                   degree-1, myTurtle)
        sierpinski([points[2],
                        getMid(points[2], points[1]),
                        getMid(points[0], points[2])],
                   degree-1, myTurtle)

def main():
   myTurtle = turtle.Turtle()
   myWin = turtle.Screen()
   myPoints = [[-100,-50],[0,100],[100,-50]]
   sierpinski(myPoints,3,myTurtle)
   myWin.exitonclick()

main()
```

*Activecode 1*

ActiveCode 1 中的程序遵循上述概念。谢尔宾斯基的第一件事是绘制外三角形。接下来，有三个递归调用，每个使我们在连接中点获得新的三角形。我们再次使用 Python 附带的 `turtle` 模块。你可以通过使用 `help('turtle')` 了解 `turtle` 可用方法的详细信息。

看下代码，想想绘制三角形的顺序。虽然三角的确切顺序取决于如何指定初始集，我们假设三角按左下，上，右下顺序。由于谢尔宾斯基函数调用自身，谢尔宾斯基以它的方式递归到左下角最小的三角形，然后开始填充其余的三角形。填充左下角顶角中的小三角形。最后，它填充在左下角中右下角的最小三角形。

有时，根据函数调用图来考虑递归算法是有帮助的。Figure 4 展示了递归调用总是向左移动。活动函数以黑色显示，非活动函数显示为灰色。向 Figure 4 底部越近，三角形越小。该功能一次完成一次绘制; 一旦它完成了绘制，它移动到左下方底部中间位置，然后继续这个过程。

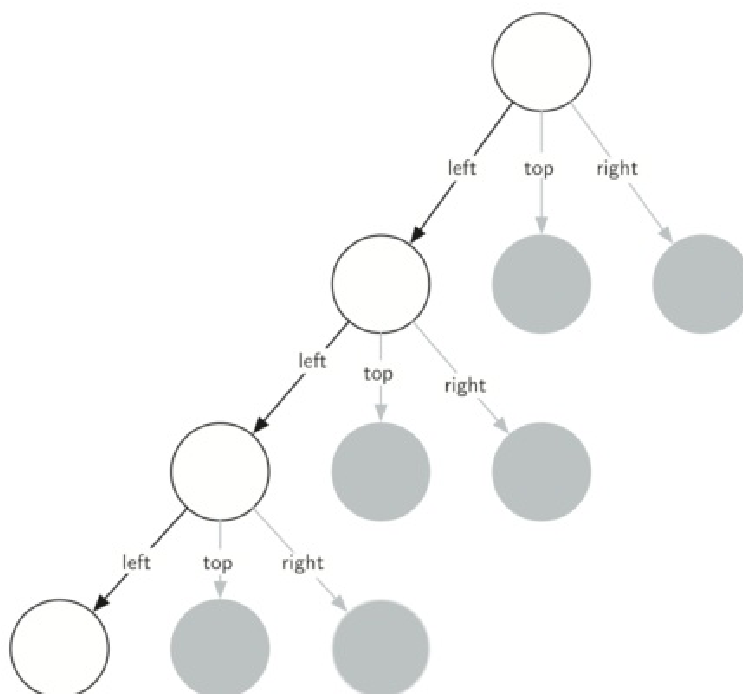



Figure 4: Building a Sierpinski Triangle

*Figure 4*

谢尔宾斯基函数在很大程度上依赖于 `getMid` 函数。 `getMid` 接受两个端点作为参数，并返回它们之间的中点。 此外，ActiveCode 1 还有一个函数，使用 `begin_fill` 和 `end_fill` 方法绘制填充一个三角形。

# 4.10.汉诺塔游戏

汉诺塔是由法国数学家爱德华·卢卡斯在 1883 年发明的。他的灵感来自一个传说，有一个印度教寺庙，将谜题交给年轻的牧师。在开始的时候，牧师们被给予三根杆和一堆 64 个金碟，每个盘比它下面一个小一点。他们的任务是将所有 64 个盘子从三个杆中一个转移到另一个。有两个重要的约束，它们一次只能移动一个盘子，并且它们不能在较小的盘子顶部上放置更大的盘子。牧师日夜不停每秒钟移动一块盘子。当他们完成工作时，传说，寺庙会变成灰尘，世界将消失。

虽然传说是有趣的，你不必担心世界不久的将来会消失。移动 64 个盘子的塔所需的步骤数是 `2^64 -1 = 18,446,744,073,709,551,615264-1 = 18,446,744,073,709,551,615` 。以每秒一次的速度，即 `584,942,417,355584,942,417,355` 年！。

Figure 1 展示了在从第一杆移动到第三杆的过程中的盘的示例。请注意，如规则指定，每个杆上的盘子都被堆叠起来，以使较小的盘子始终位于较大盘的顶部。如果你以前没有尝试过解决这个难题，你现在应该尝试下。你不需要花哨的盘子，一堆书或纸张都可以。

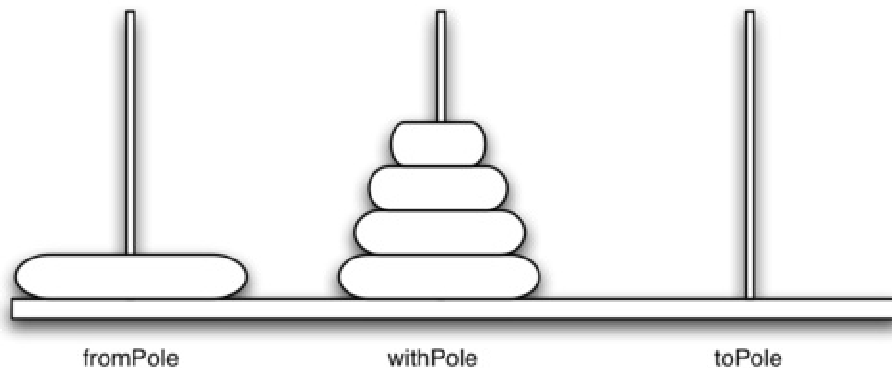



*Figure 1*

我们如何递归地解决这个问题？我们的基本情况是什么？让我们从下到上考虑这个问题。假设你有一个五个盘子的塔，在杆一上。如果你已经知道如何将四个盘子移动到杆二上，那么你可以轻松地将最底部的盘子移动到杆三，然后再将四个盘子从杆二移动到杆三。但是如果你不知道如何移动四个盘子怎么办？假设你知道如何移动三个盘子到杆三;那么很容易将第四个盘子移动到杆二，并将它们从杆三移动到它们的顶部。但是如果你不知道如何移动三个盘子呢？如何将两个盘子移动到杆二，然后将第三个盘子移动到杆三，然后移动两个盘子到它的顶部？但是如果你还不知道该怎么办呢？当然你会知道移动一个盘子到杆三足够容易。这听起来像是基本情况。

这里是如何使用中间杆将塔从起始杆移动到目标杆的步骤：

1. 使用目标杆将 `height-1` 的塔移动到中间杆。

2. 将剩余的盘子移动到目标杆。
3. 使用起始杆将 `height-1` 的塔从中间杆移动到目标杆。

只要我们遵守规则，较大的盘子保留在栈的底部，我们可以使用递归的三个步骤，处理任何更大的盘子。上面概要中唯一缺失的是识别基本情况。最简单的汉诺塔是一个盘子的塔。在这种情况下，我们只需要将一个盘子移动到其最终目的地。 一个盘子的塔将是我们的基本情况。 此外，上述步骤通过在步骤1和3中减小塔的高度，使我们趋向基本情况。Listing 1 展示了解决汉诺塔的 Python 代码。

```
def moveTower(height,fromPole, toPole, withPole):
    if height >= 1:
        moveTower(height-1,fromPole,withPole,toPole)
        moveDisk(fromPole,toPole)
        moveTower(height-1,withPole,toPole,fromPole)
```

*Listing 1*

请注意，Listing 1 中的代码与描述几乎相同。算法的简单性的关键在于我们进行两个不同的递归调用，一个在第 3 行上，另一个在第 5 行。在第 3 行上，我们将初始杆上的底部圆盘移动到中间。下一行简单地将底部盘移动到其最终的位置。然后在第 5 行上，我们将塔从中间杆移动到最大盘子的顶部。当塔高度为 0 时检测到基本情况; 在这种情况下不需要做什么，所以 `moveTower` 函数简单地返回。关于以这种方式处理基本情况的重点是，从 `moveTower` 简单地返回以使 `moveDisk` 函数被调用。

函数 `moveDisk` ，如 Listing 2 所示，非常简单。它所做的就是打印出一个盘子从一杆移动到另一杆。 如果你输入并运行 `moveTower` 程序，你可以看到它给你一个非常有效的解决方案。

```
def moveDisk(fp,tp):
    print("moving disk from",fp,"to",tp)
```

*Listing 2*

现在你已经看到了 `moveTower` 和 `moveDisk` 的代码，你可能会想知道为什么我们没有明确记录什么盘子在什么杆上的数据结构。这里有一个提示：如果你要明确地跟踪盘子，你会使用三个 Stack 对象，每个杆一个。 答案是 Python 提供了我们需要调用的隐含的栈。

# 4.11.探索迷宫

在这一节中，我们将讨论一个与扩展机器人世界相关的问题：你如何找到自己的迷宫？ 如果你在你的宿舍有一个扫地机器人（不是所有的大学生？）你希望你可以使用你在本节中学到的知识重新给它编程。 我们要解决的问题是帮助我们的乌龟在虚拟迷宫中找到出路。 迷宫问题的根源与希腊的神话有关，传说忒修斯被送入迷宫中以杀死人身牛头怪。忒修斯用了一卷线帮助他找到回去的退路，当他完成杀死野兽的任务。在我们的问题中，我们将假设我们的乌龟在迷宫中间的某处，必须找到出路。 看看 Figure 2，了解我们将在本节中做什么。

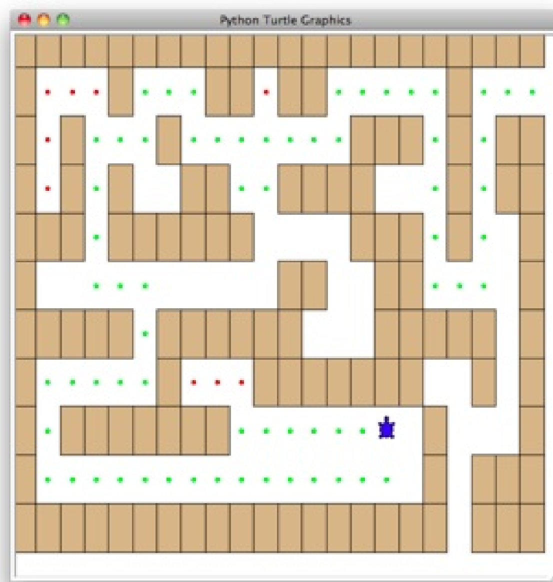



*Figure 2*

为了使问题容易些，我们假设我们的迷宫被分成“正方形”。迷宫的每个正方形是开放的或被一段墙壁占据。乌龟只能通过迷宫的空心方块。 如果乌龟撞到墙上，它必须尝试不同的方向。乌龟将需要一个程序，以找到迷宫的出路。这里是过程：

1. 从我们的起始位置，我们将首先尝试向北一格，然后从那里递归地尝试我们的程序。
2. 如果我们通过尝试向北作为第一步没有成功，我们将向南一格，并递归地重复我们的程序。
3. 如果向南也不行，那么我们将尝试向西一格，并递归地重复我们的程序。
4. 如果北，南和西都没有成功，则应用程序从当前位置递归向东。
5. 如果这些方向都没有成功，那么没有办法离开迷宫，我们失败。

现在，这听起来很容易，但有几个细节先谈谈。假设我们第一步是向北走。按照我们的程序，我们的下一步也将是向北。但如果北面被一堵墙阻挡，我们必须看看程序的下一步，并试着向南。不幸的是，向南使我们回到原来的起点。如果我们从那里再次应用递归过程，我们将又回到向北一格，并陷入无限循环。所以，我们必须有一个策略来记住我们去过哪。在

这种情况下，我们假设有一袋面包屑可以撒在我们走过的路上。如果我们沿某个方向迈出一步，发现那个位置上已经有面包屑，我们应该立即后退并尝试程序中的下一个方向。我们看看这个算法的代码，就像从递归函数调用返回一样简单。

正如我们对所有递归算法所做的一样，让我们回顾一下基本情况。其中一些你可能已经根据前一段的描述猜到了。在这种算法中，有四种基本情况要考虑：

1. 乌龟撞到了墙。由于这一格被墙壁占据，不能进行进一步的探索。
2. 乌龟找到一个已经探索过的格。我们不想继续从这个位置探索，否则会陷入循环。
3. 我们发现了一个外边缘，没有被墙壁占据。换句话说，我们发现了迷宫的一个出口。
4. 我们探索了一格在四个方向上都没有成功。

为了我们的程序工作，我们将需要有一种方式来表示迷宫。为了使这个更有趣，我们将使用 `turtle` 模块来绘制和探索我们的迷宫，以使我们看到这个算法的功能。迷宫对象将提供以下方法让我们在编写搜索算法时使用：

- `__init__` 读取迷宫的数据文件，初始化迷宫的内部表示，并找到乌龟的起始位置。
- `drawMaze` 在屏幕上的一个窗口中绘制迷宫。
- `updatePosition` 更新迷宫的内部表示，并更改窗口中乌龟的位置。
- `isExit` 检查当前位置是否是迷宫的退出位置。

Maze 类还重载索引运算符 `[]` ，以便我们的算法可以轻松访问任何特定格的状态。

让我们来查看称为 `searchFrom` 的搜索函数的代码。代码如 Listing 3 所示。请注意，此函数需要三个参数：迷宫对象，起始行和起始列。 这很重要，因为作为递归函数，搜索在每次递归调用时开始。

```
def searchFrom(maze, startRow, startColumn):
    maze.updatePosition(startRow, startColumn)
   #  Check for base cases:
   #  1. We have run into an obstacle, return false
   if maze[startRow][startColumn] == OBSTACLE :
        return False
    #  2. We have found a square that has already been explored
    if maze[startRow][startColumn] == TRIED:
        return False
    # 3. Success, an outside edge not occupied by an obstacle
    if maze.isExit(startRow,startColumn):
        maze.updatePosition(startRow, startColumn, PART_OF_PATH)
        return True
    maze.updatePosition(startRow, startColumn, TRIED)

    # Otherwise, use logical short circuiting to try each
    # direction in turn (if needed)
    found = searchFrom(maze, startRow-1, startColumn) or \
            searchFrom(maze, startRow+1, startColumn) or \
            searchFrom(maze, startRow, startColumn-1) or \
            searchFrom(maze, startRow, startColumn+1)
    if found:
        maze.updatePosition(startRow, startColumn, PART_OF_PATH)
    else:
        maze.updatePosition(startRow, startColumn, DEAD_END)
    return found
```

*Listing 3*

你会看到代码的第一行（行 2）调用 `updatePosition` 。这只是为了可视化算法，以便你可以看到乌龟如何探索通过迷宫。接下来算法检查四种基本情况中的前三种：乌龟是否碰到墙（行 5）？乌龟是否回到已经探索过的格子（行 8）？乌龟有没有到达出口（行 11）？如果这些条件都不为真，则我们继续递归搜索。

你会注意到，在递归步骤中有四个对 `searchFrom` 的递归调用。很难预测将有多少个递归调用，因为它们都由 `or` 语句连接。如果对 `searchFrom` 的第一次调用返回 `True` ，则不需要最后三个调用。你可以理解这一步向 `(row-1,column)` （或北，如果你从地理位置上思考）是在迷宫的路径上。如果没有一个好的路径向北，那么尝试下一个向南的递归调用。如果向南失败，然后尝试向西，最后向东。如果所有四个递归调用返回 `False` ，那么认为是一个死胡同。你应该下载或输入整个程序，并通过更改这些调用的顺序进行实验。

`Maze` 类的代码如 Listing 4，Listing 5 和 Listing 6 所示。 `__init__` 方法将文件的名称作为其唯一参数。此文件是一个文本文件，通过使用 `+` 字符表示墙壁，空格表示空心方块，并使用字母 `S` 表示起始位置。Figure 3 是迷宫数据文件的示例。迷宫的内部表示是列表的列表。 `mazelist` 实例变量的每一行也是一个列表。此辅助列表使用上述字符，每格表示一个字符。Figure 3 中的数据文件，内部表示如下所示：

```
[ ['+','+','+','+',...,'+','+','+','+','+','+','+'],
  ['+',' ',' ',' ',...,' ',' ',' ','+',' ',' ',' '],
  ['+',' ','+',' ',...,'+','+',' ','+',' ','+','+'],
  ['+',' ','+',' ',...,' ',' ',' ','+',' ','+','+'],
  ['+','+','+',' ',...,'+','+',' ','+',' ',' ','+'],
  ['+',' ',' ',' ',...,'+','+',' ',' ',' ',' ','+'],
  ['+','+','+','+',...,'+','+','+','+','+',' ','+'],
  ['+',' ',' ',' ',...,'+','+',' ',' ','+',' ','+'],
  ['+',' ','+','+',...,' ',' ','+',' ',' ',' ','+'],
  ['+',' ',' ',' ',...,' ',' ','+',' ','+','+','+'],
  ['+','+','+','+',...,'+','+','+',' ','+','+','+']]
```

`drawMaze` 方法使用这个内部表示在屏幕上绘制迷宫的初始视图。

Figure 3：示例迷宫数据文件

```
++++++++++++++++++++++
+   +   ++ ++     +
+ +   +       +++ + ++
+ + +  ++  ++++   + ++
+++ ++++++    +++ +  +
+          ++  ++    +
+++++ ++++++    +++++ +
+     +   +++++++  + +
+ +++++++      S +   +
+                + +++
++++++++++++++++++ +++
```

*Figure 3*

如 Listing 5 所示，`updatePosition` 方法使用相同的内部表示来查看乌龟是否遇到了墙。它还用 `.` 或 `-` 更新内部表示，以表示乌龟已经访问了特定格子或者格子是死角。 此外，`updatePosition` 方法使用两个辅助方法`moveTurtle` 和 `dropBreadCrumb` 来更新屏幕上的视图。

最后，`isExit` 方法使用乌龟的当前位置来检测退出条件。退出条件是当乌龟已经到迷宫的边缘时，即行零或列零，或者在最右边列或底部行。

```
class Maze:
    def __init__(self,mazeFileName):
        rowsInMaze = 0
        columnsInMaze = 0
        self.mazelist = []
        mazeFile = open(mazeFileName,'r')
        rowsInMaze = 0
        for line in mazeFile:
            rowList = []
            col = 0
            for ch in line[:-1]:
                rowList.append(ch)
                if ch == 'S':
                    self.startRow = rowsInMaze
                    self.startCol = col
                col = col + 1
            rowsInMaze = rowsInMaze + 1
            self.mazelist.append(rowList)
            columnsInMaze = len(rowList)

        self.rowsInMaze = rowsInMaze
        self.columnsInMaze = columnsInMaze
        self.xTranslate = -columnsInMaze/2
        self.yTranslate = rowsInMaze/2
        self.t = Turtle(shape='turtle')
        setup(width=600,height=600)
        setworldcoordinates(-(columnsInMaze-1)/2-.5,
                            -(rowsInMaze-1)/2-.5,
                            (columnsInMaze-1)/2+.5,
                            (rowsInMaze-1)/2+.5)
```

*Listing 4*

```
def drawMaze(self):
    for y in range(self.rowsInMaze):
        for x in range(self.columnsInMaze):
            if self.mazelist[y][x] == OBSTACLE:
                self.drawCenteredBox(x+self.xTranslate,
                                     -y+self.yTranslate,
                                     'tan')
    self.t.color('black','blue')

def drawCenteredBox(self,x,y,color):
    tracer(0)
    self.t.up()
    self.t.goto(x-.5,y-.5)
    self.t.color('black',color)
    self.t.setheading(90)
    self.t.down()
    self.t.begin_fill()
    for i in range(4):
        self.t.forward(1)
        self.t.right(90)
    self.t.end_fill()
    update()
    tracer(1)

def moveTurtle(self,x,y):
    self.t.up()
    self.t.setheading(self.t.towards(x+self.xTranslate,
                                     -y+self.yTranslate))
    self.t.goto(x+self.xTranslate,-y+self.yTranslate)

def dropBreadcrumb(self,color):
    self.t.dot(color)

def updatePosition(self,row,col,val=None):
    if val:
        self.mazelist[row][col] = val
    self.moveTurtle(col,row)

    if val == PART_OF_PATH:
        color = 'green'
    elif val == OBSTACLE:
        color = 'red'
    elif val == TRIED:
        color = 'black'
    elif val == DEAD_END:
        color = 'red'
    else:
        color = None

    if color:
        self.dropBreadcrumb(color)
```

*Listing 5*

```
def isExit(self,row,col):
     return (row == 0 or
             row == self.rowsInMaze-1 or
             col == 0 or
             col == self.columnsInMaze-1 )

def __getitem__(self,idx):
     return self.mazelist[idx]
```

*Listing 6*

# 4.12.动态规划

计算机科学中的许多程序是为了优化一些值而编写的; 例如，找到两个点之间的最短路径，找到最适合一组点的线，或找到满足某些标准的最小对象集。计算机科学家使用许多策略来解决这些问题。本书的目标之一是向你展示几种不同的解决问题的策略。 `动态规划` 是这些类型的优化问题的一个策略。

优化问题的典型例子包括使用最少的硬币找零。假设你是一个自动售货机制造商的程序员。你的公司希望通过给每个交易最少硬币来简化工作。假设客户放入 1 美元的钞票并购买 37 美分的商品。你可以用来找零的最小数量的硬币是多少？答案是六个硬币：两个 25 美分，一个 10美分 和 三个一美分。我们如何得到六个硬币的答案？我们从最大的硬币（25 美分）开始，并尽可能多，然后我们去找下一个小点的硬币，并尽可能多的使用它们。这第一种方法被称为贪婪方法，因为我们试图尽快解决尽可能大的问题。

当我们使用美国货币时，贪婪的方法工作正常，但是假设你的公司决定在埃尔博尼亚部署自动贩卖机，除了通常的 1，5，10 和 25 分硬币，他们还有一个 21 分硬币 。在这种情况下，我们的贪婪的方法找不到 63 美分的最佳解决方案。 随着加入 21分硬币，贪婪的方法仍然会找到解决方案是六个硬币。然而，最佳答案是三个 21 分。

让我们看一个方法，我们可以确定会找到问题的最佳答案。由于这一节是关于递归的，你可能已经猜到我们将使用递归解决方案。让我们从基本情况开始，如果我们可以与我们硬币的价值相同的金额找零，答案很容易，一个硬币。

如果金额不匹配，我们有几个选项。我们想要的是最低一个一分钱加上原始金额减去一分钱所需的硬币数量，或者一个 5 美分加上原始金额减去 5 美分所需的硬币数量，或者一个 10 美分加上原始金额减去 10 美分所需的硬币数量，等等。因此，需要对原始金额找零硬币数量可以根据下式计算：

$$numCoins = min\begin{cases} 1 + numCoins(originalamount - 1) \\ 1 + numCoins(originalamount - 5) \\ 1 + numCoins(originalamount - 10) \\ 1 + numCoins(originalamount - 25) \end{cases}$$

执行我们刚才描述的算法如 Listing 7 所示。在第 3 行，我们检查基本情况;也就是说，我们正试图支付硬币的确切金额。如果我们没有等于找零数量的硬币，我们递归调用每个小于找零额的不同的硬币值。第 6 行显示了我们如何使用列表推导将硬币列表过滤到小于当前找零的硬币列表。递归调用也减少了由所选硬币的值所需要的总找零量。递归调用在第 7 行。注意在同一行，我们将硬币数 `+1` ，以说明我们正在使用一个硬币的事实。

```
def recMC(coinValueList,change):
   minCoins = change
   if change in coinValueList:
     return 1
   else:
      for i in [c for c in coinValueList if c <= change]:
         numCoins = 1 + recMC(coinValueList,change-i)
         if numCoins < minCoins:
            minCoins = numCoins
   return minCoins

print(recMC([1,5,10,25],63))
```

*Listing 7*

Listing 7 中的算法是非常低效的。事实上，它需要 `67,716,925` 个递归调用来找到 4 个硬币的最佳解决 63 美分问题的方案。要理解我们方法中的致命缺陷，请参见 Figure 5，其中显示了 377 个函数调用所需的一小部分，找到支付 26 美分的最佳硬币。

图中的每个节点对应于对 `recMC` 的调用。节点上的标签表示硬币数量的变化量。箭头上的标签表示我们刚刚使用的硬币。通过跟随图表，我们可以看到硬币的组合。主要的问题是我们重复做了太多的计算。例如，该图表示该算法重复计算了至少三次支付 15 美分。这些计算找到 15 美分的最佳硬币数量的步骤本身需要52个函数调用。显然，我们浪费了大量的时间和精力重新计算旧的结果。

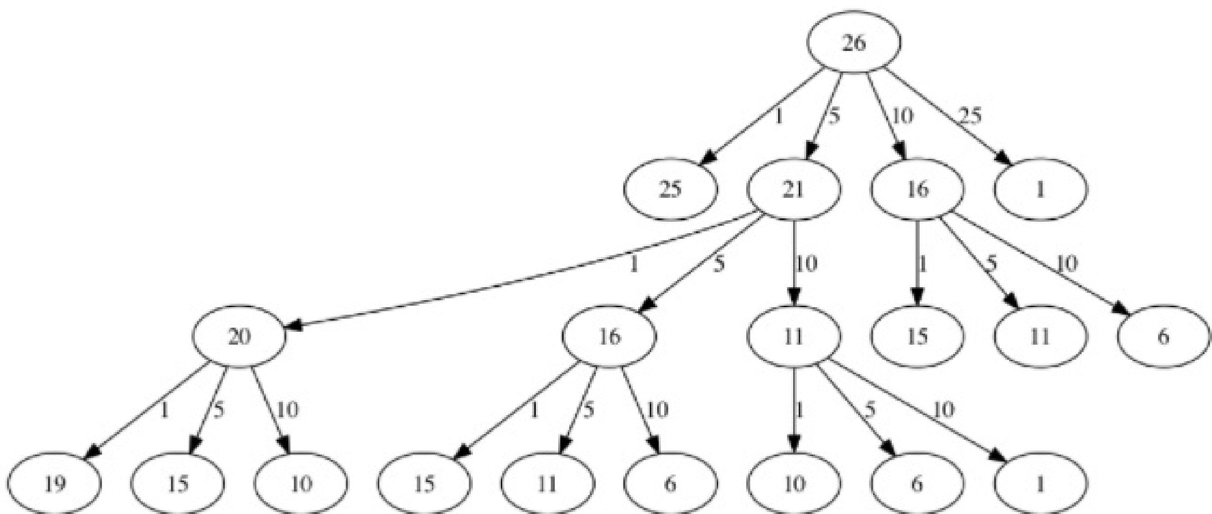


*Figure 5*

减少我们工作量的关键是记住一些过去的结果，这样我们可以避免重新计算我们已经知道的结果。一个简单的解决方案是将最小数量的硬币的结果存储在表中。然后在计算新的最小值之前，我们首先检查表，看看结果是否已知。如果表中已有结果，我们使用表中的值，而不是重新计算。 ActiveCode 1 显示了一个修改的算法，以合并我们的表查找方案。

```
def recDC(coinValueList,change,knownResults):
   minCoins = change
   if change in coinValueList:
      knownResults[change] = 1
      return 1
   elif knownResults[change] > 0:
      return knownResults[change]
   else:
       for i in [c for c in coinValueList if c <= change]:
         numCoins = 1 + recDC(coinValueList, change-i,
                              knownResults)
         if numCoins < minCoins:
            minCoins = numCoins
            knownResults[change] = minCoins
   return minCoins

print(recDC([1,5,10,25],63,[0]*64))
```

*ActiveCode 1*

注意，在第 6 行中，我们添加了一个测试，看看我们的列表是否包含此找零的最小硬币数量。如果没有，我们递归计算最小值，并将计算出的最小值存储在列表中。使用这个修改的算法减少了我们需要为四个硬币递归调用的数量，63美分问题只需 221 次调用！

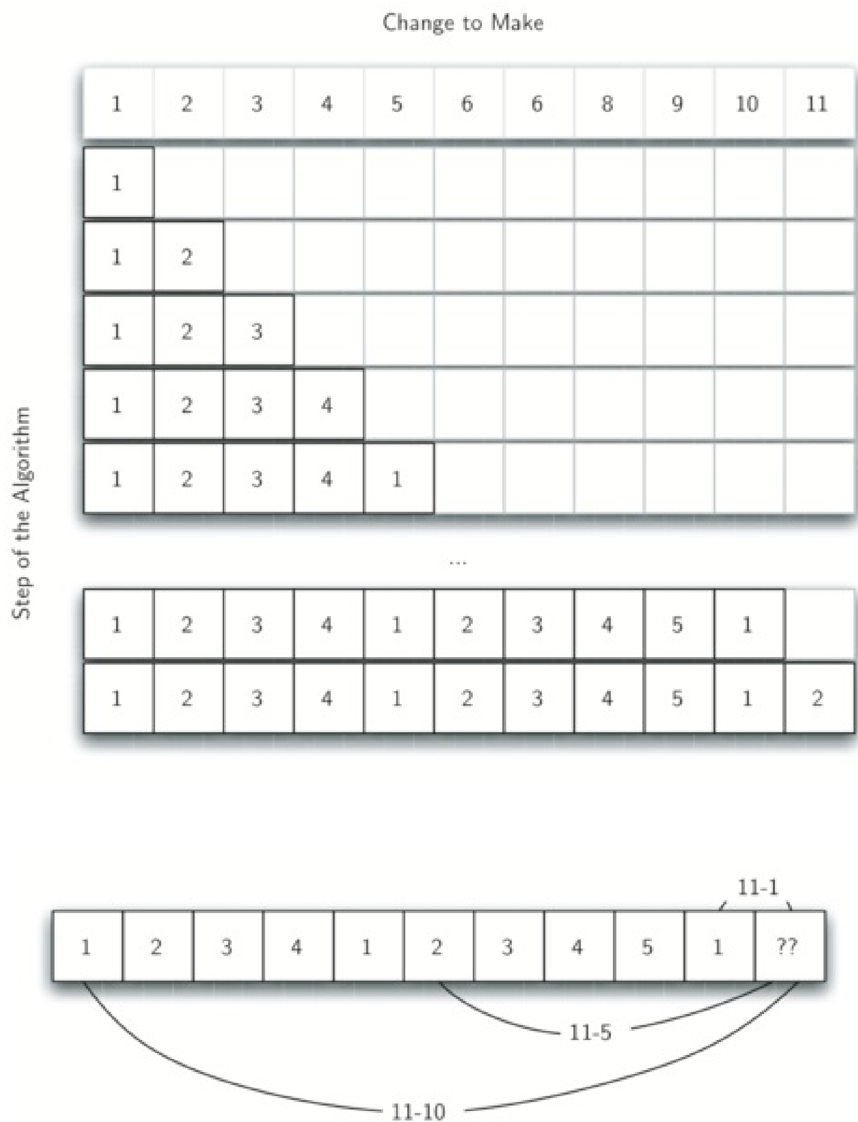
虽然 AcitveCode 1 中的算法是正确的。事实上，我们所做的不是动态规划，而是我们通过使用称为 `记忆化` 的技术来提高我们的程序的性能，或者更常见的叫做 `缓存` 。

一个真正的动态编程算法将采取更系统的方法来解决这个问题。我们的动态编程解决方案将从找零一分钱开始，并系统地找到我们需要的找零额。这保证我们在算法的每一步，已经知道为任何更小的数量进行找零所需的最小硬币数量。

让我们看看如何找到 11 美分所需的最小找零数量。Figure 4 说明了该过程。我们从一分钱开始。唯一的解决方案是一个硬币（一分钱）。下一行显示一分和两分的最小值。再次，唯一的解决方案是两分钱。第五行事情变得有趣。现在我们有两个选择，五个一分钱或一个五分钱。我们如何决定哪个是最好的？我们查阅表，看到需要找零四美分的硬币数量是四，再加一个一分钱是五，等于五个硬币。或者我们可以尝试 0 分加一个五分，五分钱等于一个硬币。由于一和五最小的是一，我们在表中存储为一。再次快进到表的末尾，考虑 11 美分。Figure 5 展示了我们要考虑的三个选项：

1. 一个一分钱加上 `11-1 = 10分（1）` 的最小硬币数
2. 一个五分钱加上 `11-5 = 6分（2）` 的最小硬币数
3. 一个十分钱加上 `11-10 = 1 分（1）` 最小硬币数

选项 1 或 3 总共需要两个硬币，这是 11 美分的最小硬币数。




Listing 8 用一个动态规划算法来解决我们的找零问题。 `dpMakeChange` 有三个参数：一个有效硬币值的列表，我们要求的找零额，以及一个包含每个值所需最小硬币数量的列表。 当函数完成时， `minCoins` 将包含从 0 到找零值的所有值的解。

```
def dpMakeChange(coinValueList,change,minCoins):
   for cents in range(change+1):
      coinCount = cents
      for j in [c for c in coinValueList if c <= cents]:
            if minCoins[cents-j] + 1 < coinCount:
               coinCount = minCoins[cents-j]+1
      minCoins[cents] = coinCount
   return minCoins[change]
```

*Listing 8*

注意，`dpMakeChange` 不是递归函数，即使我们开始使用递归解决这个问题。重要的是要意识到，你可以为问题写一个递归解决方案但并不意味着它是最好的或最有效的解决方案。在这个函数中的大部分工作是通过从第 4 行开始的循环来完成的。在这个循环中，我们考虑使用所有可能的硬币对指定的金额进行找零。就像我们上面的 11 分的例子，我们记住最小值，并将其存储在我们的 `minCoins` 列表。

虽然我们的找零算法很好地找出最小数量的硬币，但它不帮助我们找零，因为我们不跟踪我们使用的硬币。我们可以轻松地扩展 `dpMakeChange` 来跟踪硬币使用，只需记住我们为每个条目添加的最后一个硬币到 `minCoins` 表。如果我们知道添加的最后一个硬币值，我们可以简单地减去硬币的值，在表中找到前一个条目，找到该金额的最后一个硬币。我们可以通过表继续跟踪，直到我们开始的位置。

ActiveCode 2 展示了 `dpMakeChange` 算法修改为跟踪使用的硬币，以及一个函数 `printCoins` 通过表打印出使用的每个硬币的值。前两行主要设置要找零的金额，并创建使用的硬币列表。 接下来的两行创建了我们需要存储结果的列表。`coinsUsed` 是用于找零的硬币的列表，并且 `coinCount` 是与列表中的位置相对应进行找零的最小硬币数。

注意，我们打印的硬币直接来自 `coinsUsed` 数组。对于第一次调用，我们从数组位置 `63` 开始，然后打印 `21` 。然后我们取 `63-21 = 42` ，看看列表的第 42 个元素。我们再次找到 21 存储在那里。 最后，数组第 21 个元素21 也包含 21，得到三个 21。

```
def dpMakeChange(coinValueList,change,minCoins,coinsUsed):
   for cents in range(change+1):
      coinCount = cents
      newCoin = 1
      for j in [c for c in coinValueList if c <= cents]:
            if minCoins[cents-j] + 1 < coinCount:
               coinCount = minCoins[cents-j]+1
               newCoin = j
      minCoins[cents] = coinCount
      coinsUsed[cents] = newCoin
   return minCoins[change]

def printCoins(coinsUsed,change):
   coin = change
   while coin > 0:
      thisCoin = coinsUsed[coin]
      print(thisCoin)
      coin = coin - thisCoin

def main():
    amnt = 63
    clist = [1,5,10,21,25]
    coinsUsed = [0]*(amnt+1)
    coinCount = [0]*(amnt+1)

    print("Making change for",amnt,"requires")
    print(dpMakeChange(clist,amnt,coinCount,coinsUsed),"coins")
    print("They are:")
    printCoins(coinsUsed,amnt)
    print("The used list is as follows:")
    print(coinsUsed)

main()
```

*AcitveCode 2*

```
Making change for 63 requires
3 coins
They are:
21
21
21
The used list is as follows:
[1, 1, 1, 1, 1, 5, 1, 1, 1, 1, 10, 1, 1, 1, 1, 5, 1, 1, 1, 1, 10, 21, 1, 1, 1, 25, 1,
1, 1, 1, 5, 10, 1, 1, 1, 10, 1, 1, 1, 1, 5, 10, 21, 1, 1, 10, 21, 1, 1, 1, 25, 1, 10,
1, 1, 5, 10, 1, 1, 1, 10, 1, 10, 21]
```

# 4.13.总结

在本章中，我们讨论了几个递归算法的例子。 选择这些算法来揭示几个不同的问题，其中递归是一种有效的问题解决技术。 本章要记住的要点如下：

- 所有递归算法都必须具有基本情况。
- 递归算法必须改变其状态并朝基本情况发展。
- 递归算法必须调用自身（递归）。
- 递归在某些情况下可以代替迭代。
- 递归算法通常可以自然地映射到你尝试解决的问题的表达式。
- 递归并不总是答案。有时，递归解决方案可能比迭代算法在计算上更昂贵。

# 5.1. 目标

- 能够解释和实现顺序查找和二分查找。
- 能够解释和实现选择排序，冒泡排序，归并排序，快速排序，插入排序和 shell 排序。
- 理解哈希作为搜索技术的思想。
- 引入映射抽象数据类型。
- 使用哈希实现 Map 抽象数据类型。

# 5.2.搜索

我们现在把注意力转向计算中经常出现的一些问题，即搜索和排序问题。在本节中，我们将研究搜索。我们将在本章后面的章节中介绍。搜索是在项集合中查找特定项的算法过程。搜索通常对于项是否存在返回 True 或 False。有时它可能返回项被找到的地方。我们在这里将仅关注成员是否存在这个问题。

在 Python 中，有一个非常简单的方法来询问一个项是否在一个项列表中。我们使用 in 运算符。

```
>>> 15 in [3,5,2,4,1]
False
>>> 3 in [3,5,2,4,1]
True
>>>
```

这很容易写，一个底层的操作替我们完成这个工作。事实证明，有很多不同的方法来搜索。我们在这里感兴趣的是这些算法如何工作以及它们如何相互比较。

# 5.3.顺序查找

当数据项存储在诸如列表的集合中时，我们说它们具有线性或顺序关系。 每个数据项都存储在相对于其他数据项的位置。 在 Python 列表中，这些相对位置是单个项的索引值。由于这些索引值是有序的，我们可以按顺序访问它们。 这个过程产生我们的第一种搜索技术 `顺序查找` 。

Figure 1 展示了这种搜索的工作原理。 从列表中的第一个项目开始，我们按照基本的顺序排序，简单地从一个项移动到另一个项，直到找到我们正在寻找的项或遍历完整个列表。如果我们遍历完整个列表，则说明正在搜索的项不存在。

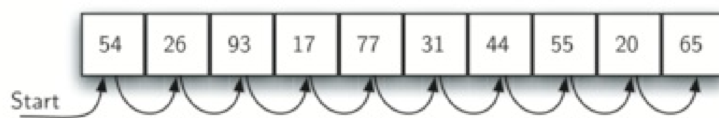



*Figure 1*

该算法的 Python 实现见 CodeLens 1。该函数需要一个列表和我们正在寻找的项作为参数，并返回一个是否存在的布尔值。 `found` 布尔变量初始化为 False，如果我们发现列表中的项，则赋值为 True。

```
def sequentialSearch(alist, item):
        pos = 0
        found = False

        while pos < len(alist) and not found:
            if alist[pos] == item:
                found = True
            else:
                pos = pos+1

        return found

testlist = [1, 2, 32, 8, 17, 19, 42, 13, 0]
print(sequentialSearch(testlist, 3))
print(sequentialSearch(testlist, 13))
```

*CodeLens 1*

## 5.3.1.顺序查找分析

为了分析搜索算法，我们需要定一个基本计算单位。回想一下，这通常是为了解决问题要重复的共同步骤。对于搜索，计算比较操作数是有意义的。每个比较都有可能找到我们正在寻找的项目。此外，我们在这里做另一个假设。项列表不以任何方式排序。项随机放置到列表中。换句话说，项在列表任何位置的概率是一样的。

如果项不在列表中，知道它的唯一方法是将其与存在的每个项进行比较。如果有n 个项，则顺序查找需要 n 个比较来发现项不存在。在项在列表中的情况下，分析不是那么简单。实际上有三种不同的情况可能发生。在最好的情况下，我们在列表的开头找到所需的项，只需要一个比较。在最坏的情况下，我们直到最后的比较才找到项，第 n 个比较。

平均情况怎么样？平均来说，我们会在列表的一半找到该项; 也就是说，我们将比较 n/2 项。然而，回想一下，当 n 变大时，系数，无论它们是什么，在我们的近似中变得不重要，因此顺序查找的复杂度是 O(n)。Table 1 总结了这些结果。

| Case | Best Case | Worst Case | Average Case |
|---|---|---|---|
| item is present | $1$ | $n$ | $\frac{n}{2}$ |
| item is not present | $n$ | $n$ | $n$ |

*Table 1*

我们之前假设，我们列表中的项是随机放置的，因此在项之间没有相对顺序。如果项以某种方式排序，顺序查找会发生什么？我们能够在搜索技术中取得更好的效率吗？

假设项的列表按升序排列。如果我们正在寻找的项存在于列表中，它在 n 个位置中的概率依旧相同。我们仍然会有相同数量的比较来找到该项。然而，如果该项不存在，则有一些优点。Figure 2 展示了这个过程，寻找项 50。注意，项仍然按顺序进行比较直到 54。此时，我们知道一些额外的东西。不仅 54 不是我们正在寻找的项，也没有超过 54 的其他元素可以匹配到该项，因为列表是有序的。在这种情况下，算法不必继续查看所有项。它可以立即停止。 CodeLens 2 展示了顺序查找功能的这种变化。

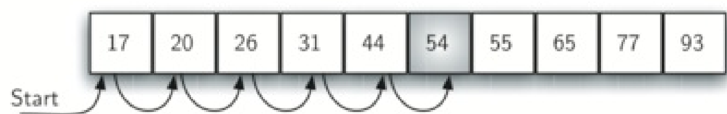



*Figure 2*

```
def orderedSequentialSearch(alist, item):
        pos = 0
        found = False
        stop = False
        while pos < len(alist) and not found and not stop:
            if alist[pos] == item:
                found = True
            else:
                if alist[pos] > item:
                    stop = True
                else:
                    pos = pos+1

        return found

testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(orderedSequentialSearch(testlist, 3))
print(orderedSequentialSearch(testlist, 13))
```

*CodeLens 2*

Table 2 总结了这些结果。 请注意，在最好的情况下，我们通过只查看一项会发现该项不在列表中。 平均来说，我们将只了解 n/2 项就知道。然而，这种复杂度仍然是 O(n)。 总之，只有在我们没有找到该项的情况下，才通过对列表排序来改进顺序查找。

| | | | |
|---|---|---|---|
| item is present | $1$ | $n$ | $\frac{n}{2}$ |
| item not present | $1$ | $n$ | $\frac{n}{2}$ |

*Table 2*

# 5.4.二分查找

有序列表对于我们的比较是很有用的。在顺序查找中，当我们与第一个项进行比较时，如果第一个项不是我们要查找的，则最多还有 `n-1` 个项目。 二分查找从中间项开始，而不是按顺序查找列表。 如果该项是我们正在寻找的项，我们就完成了查找。 如果它不是，我们可以使用列表的有序性质来消除剩余项的一半。如果我们正在查找的项大于中间项，就可以消除中间项以及比中间项小的一半元素。如果该项在列表中，肯定在大的那半部分。

然后我们可以用大的半部分重复这个过程。从中间项开始，将其与我们正在寻找的内容进行比较。再次，我们找到元素或将列表分成两半，消除可能的搜索空间的另一部分。Figure 3 展示了该算法如何快速找到值 54 。完整的函数见CodeLens 3中。

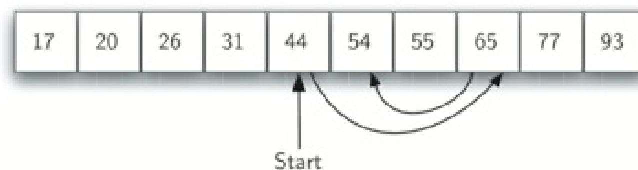



*Figure 3*

```
def binarySearch(alist, item):
        first = 0
        last = len(alist)-1
        found = False

        while first<=last and not found:
            midpoint = (first + last)//2
            if alist[midpoint] == item:
                found = True
            else:
                if item < alist[midpoint]:
                    last = midpoint-1
                else:
                    first = midpoint+1

        return found

testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(binarySearch(testlist, 3))
print(binarySearch(testlist, 13))
```

*CodeLens 3*

在我们继续分析之前，我们应该注意到，这个算法是分而治之策略的一个很好的例子。分和治意味着我们将问题分成更小的部分，以某种方式解决更小的部分，然后重新组合整个问题以获得结果。 当我们执行列表的二分查找时，我们首先检查中间项。如果我们正在搜索的项小于中间项，我们可以简单地对原始列表的左半部分执行二分查找。同样，如果项大，我们可以执行右半部分的二分查找。 无论哪种方式，都是递归调用二分查找函数。 CodeLens 4 展示了这个递归版本。

```
def binarySearch(alist, item):
        if len(alist) == 0:
            return False
        else:
            midpoint = len(alist)//2
            if alist[midpoint]==item:
              return True
            else:
              if item<alist[midpoint]:
                return binarySearch(alist[:midpoint],item)
              else:
                return binarySearch(alist[midpoint+1:],item)

testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(binarySearch(testlist, 3))
print(binarySearch(testlist, 13))
```

*CodeLens 4*

## 5.4.1.二分查找分析

为了分析二分查找算法，我们需要记住，每个比较消除了大约一半的剩余项。该算法检查整个列表的最大比较数是多少？如果我们从 n 项开始，大约 n/2 项将在第一次比较后留下。第二次比较后，会有约 n/4。 然后 n/8，n/16，等等。 我们可以拆分列表多少次？ Table 3 帮助我们找到答案。

| Comparisons | Approximate Number of Items Left |
| --- | --- |
| 1 | $\frac{n}{2}$ |
| 2 | $\frac{n}{4}$ |
| 3 | $\frac{n}{8}$ |
| ... | |
| i | $\frac{n}{2^i}$ |

*Table 3*

当我们切分列表足够多次时，我们最终得到只有一个项的列表。 要么是我们正在寻找的项，要么不是。达到这一点所需的比较数是 i，当 n/2^i = 1 时。 求解 i 得出 i = log^n 。 最大比较数相对于列表中的项是对数的。 因此，二分查找是 O( log^n )。

还需要解决一个额外的分析问题。在上面所示的递归解中，递归调用，

```
binarySearch(alist[:midpoint],item)
```

使用切片运算符创建列表的左半部分，然后传递到下一个调用（同样对于右半部分）。我们上面做的分析假设切片操作符是恒定时间的。然而，我们知道 Python中的 slice 运算符实际上是 O(k)。这意味着使用 slice 的二分查找将不会在严格的对数时间执行。幸运的是，这可以通过传递列表连同开始和结束索引来纠正。可以像 CodeLens 3 中所做的那样计算索引。我们将此实现作为练习。

即使二分查找通常比顺序查找更好，但重要的是要注意，对于小的 n 值，排序的额外成本可能不值得。事实上，我们应该经常考虑采取额外的分类工作是否使搜索获得好处。如果我们可以排序一次，然后查找多次，排序的成本就不那么重要。然而，对于大型列表，一次排序可能是非常昂贵，从一开始就执行顺序查找可能是最好的选择。

# 5.5.Hash查找

在前面的部分中，我们通过利用关于项在集合中相对于彼此存储的位置的信息，改进我们的搜索算法。例如，通过知道列表是有序的，我们可以使用二分查找在对数时间中搜索。在本节中，我们将尝试进一步建立一个可以在 O(1) 时间内搜索的数据结构。这个概念被称为 `Hash 查找` 。

为了做到这一点，当我们在集合中查找项时，我们需要更多地了解项可能在哪里。如果每个项都在应该在的地方，那么搜索可以使用单个比较就能发现项的存在。然而，我们看到，通常不是这样的。

`哈希表` 是以一种容易找到它们的方式存储的项的集合。哈希表的每个位置，通常称为一个槽，可以容纳一个项，并且由从 0 开始的整数值命名。例如，我们有一个名为 0 的槽，名为 1 的槽，名为 2 的槽，以上。最初，哈希表不包含项，因此每个槽都为空。我们可以通过使用列表来实现一个哈希表，每个元素初始化为 `None` 。Figure 4 展示了大小 m = 11 的哈希表。换句话说，在表中有 m 个槽，命名为 0 到 10。

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| None | None | None | None | None | None | None | None | None | None | None |

*Figure 4*

项和该项在散列表中所属的槽之间的映射被称为 `hash 函数` 。 hash 函数将接收集合中的任何项，并在槽名范围内（0和 m-1之间）返回一个整数。假设我们有整数项 `54,26,93,17,77` 和 `31` 的集合。我们的第一个 hash 函数，有时被称为 `余数法` ，只需要一个项并将其除以表大小，返回剩余部分作为其散列值 `(h(item) = item%11)` 。 Table 4 给出了我们的示例项的所有哈希值。注意，这种余数方法（模运算）通常以某种形式存在于所有散列函数中，因为结果必须在槽名的范围内。

| Item | Hash Value |
| --- | --- |
| 54 | 10 |
| 26 | 4 |
| 93 | 5 |
| 17 | 6 |
| 77 | 0 |
| 31 | 9 |

*Table 4*

一旦计算了哈希值，我们可以将每个项插入到指定位置的哈希表中，如 Figure 5 所示。注意，11 个插槽中的 6 个现在已被占用。这被称为负载因子，通常表示为 `λ=项数/表大小` , 在这个例子中， `λ = 6/11` 。

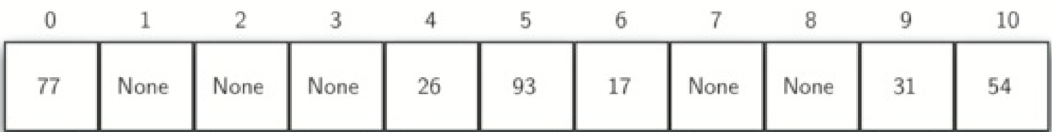



*Figure 5*

现在，当我们要搜索一个项时，我们只需使用哈希函数来计算项的槽名称，然后检查哈希表以查看它是否存在。该搜索操作是 O(1)，因为需要恒定的时间量来计算散列值，然后在该位置索引散列表。如果一切都正确的话，我们已经找到了一个恒定时间搜索算法。

你可能已经看到，只有每个项映射到哈希表中的唯一位置，这种技术才会起作用。 例如，如果项 44 是我们集合中的下一个项，则它的散列值为 `0（44%11 == 0）` 。 因为 77 的哈希值也是 0，我们会有一个问题。根据散列函数，两个或更多项将需要在同一槽中。这种现象被称为碰撞（它也可以被称为“冲突”）。显然，冲突使散列技术产生了问题。我们将在后面详细讨论。

## 5.5.1.hash 函数

给定项的集合，将每个项映射到唯一槽的散列函数被称为完美散列函数。如果我们知道项和集合将永远不会改变，那么可以构造一个完美的散列函数。不幸的是，给定任意的项集合，没有系统的方法来构建完美的散列函数。幸运的是，我们不需要散列函数是完美的，仍然可以提高性能。

总是具有完美散列函数的一种方式是增加散列表的大小，使得可以容纳项范围中的每个可能值。这保证每个项将具有唯一的槽。虽然这对于小数目的项是实用的，但是当可能项的数目大时是不可行的。例如，如果项是九位数的社保号码，则此方法将需要大约十亿个槽。如果我们只想存储 25 名学生的数据，我们将浪费大量的内存。

我们的目标是创建一个散列函数，最大限度地减少冲突数，易于计算，并均匀分布在哈希表中的项。有很多常用的方法来扩展简单余数法。我们将在这里介绍其中几个。

`分组求和法` 将项划分为相等大小的块（最后一块可能不是相等大小）。然后将这些块加在一起以求出散列值。例如，如果我们的项是电话号码 `436-555-4601` ，我们将取出数字，并将它们分成2位数 `（43,65,55,46,01）` 。 `43 + 65 + 55 + 46 + 01` ，我们得到 210。我们假设哈希表有 11 个槽，那么我们需要除以 11 。在这种情况下， `210%11` 为 1，因此电话号码 `436-555-4601` 散列到槽 1 。一些分组求和法会在求和之前每隔一个反转。对于上述示例，我们得到 `43 + 56 + 55 + 64 + 01 = 219` ，其给出 `219%11 = 10` 。

用于构造散列函数的另一数值技术被称为 `平方取中法` 。我们首先对该项平方，然后提取一部分数字结果。例如，如果项是 44，我们将首先计算 `44^2 = 1,936` 。通过提取中间两个数字 `93` ，我们得到 `5（93%11）` 。Table 5 展示了余数法和中间平方法下的项。

| Item | Remainder | Mid-Square |
| --- | --- | --- |
| 54 | 10 | 3 |
| 26 | 4 | 7 |
| 93 | 5 | 9 |
| 17 | 6 | 8 |
| 77 | 0 | 4 |
| 31 | 9 | 6 |

*Table 5*

我们还可以为基于字符的项（如字符串）创建哈希函数。 词 `cat` 可以被认为是 ascii 值的序列。

```
>>> ord('c')
99
>>> ord('a')
97
>>> ord('t')
116
```

然后，我们可以获取这三个 ascii 值，将它们相加，并使用余数方法获取散列值（参见 Figure 6）。 Listing 1 展示了一个名为 hash 的函数，它接收字符串和表大小 作为参数，并返回从 `0` 到 `tablesize-1` 的范围内的散列值。

| Item | Remainder | Mid-Square |
|---|---|---|
| 54 | 10 | 3 |
| 26 | 4 | 7 |
| 93 | 5 | 9 |
| 17 | 6 | 8 |
| 77 | 0 | 4 |
| 31 | 9 | 6 |

*Figure 6*

```
def hash(astring, tablesize):
    sum = 0
    for pos in range(len(astring)):
        sum = sum + ord(astring[pos])

    return sum%tablesize
```

*Listing 1*

有趣的是，当使用此散列函数时，字符串总是返回相同的散列值。 为了弥补这一点，我们可以使用字符的位置作为权重。 Figure 7 展示了使用位置值作为加权因子的一种可能的方式。

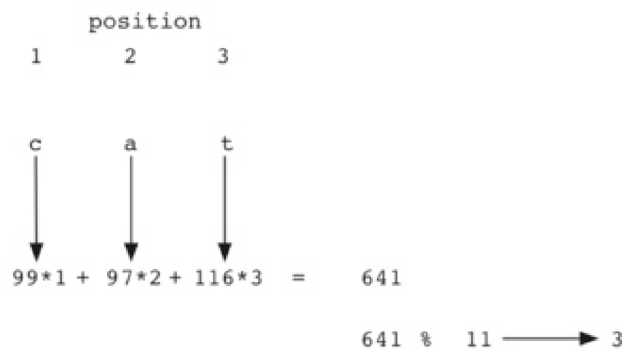



*Figure 7*

你可以思考一些其他方法来计算集合中项的散列值。重要的是要记住，哈希函数必须是高效的，以便它不会成为存储和搜索过程的主要部分。如果哈希函数太复杂，则计算槽名称的程序要比之前所述的简单地进行基本的顺序或二分搜索更耗时。 这将打破散列的目的。

## 5.5.2.冲突解决

我们现在回到碰撞的问题。当两个项散列到同一个槽时，我们必须有一个系统的方法将第二个项放在散列表中。这个过程称为冲突解决。如前所述，如果散列函数是完美的，冲突将永远不会发生。然而，由于这通常是不可能的，所以冲突解决成为散列非常重要的部分。

解决冲突的一种方法是查找散列表，尝试查找到另一个空槽以保存导致冲突的项。一个简单的方法是从原始哈希值位置开始，然后以顺序方式移动槽，直到遇到第一个空槽。注意，我们可能需要回到第一个槽（循环）以查找整个散列表。这种冲突解决过程被称为开放寻址，因为它试图在散列表中找到下一个空槽或地址。通过系统地一次访问每个槽，我们执行称为线性探测的开放寻址技术。

Figure 8展示了在简单余数法散列函数 `(54,26,93,17,77,31,44,55,20)` 下的整数项的扩展集合。上面的 Table 4 展示了原始项的哈希值。Figure 5 展示了原始内容。当我们尝试将 `44` 放入槽 0 时，发生冲突。在线性探测下，我们逐个顺序观察，直到找到位置。在这种情况下，我们找到槽 1。

再次， `55` 应该在槽 0 中，但是必须放置在槽 2 中，因为它是下一个开放位置。值 20 散列到槽 9 。由于槽 9 已满，我们进行线性探测。我们访问槽 `10,0,1` 和 `2` ，最后在位置 3 找到一个空槽。

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 77 | 44 | 55 | 20 | 26 | 93 | 17 | None | None | 31 | 54 |

*Figure 8*

一旦我们使用开放寻址和线性探测建立了哈希表，我们就必须使用相同的方法来搜索项。假设我们想查找项 `93` 。当我们计算哈希值时，我们得到 `5` 。查看槽 5 得到 `93` ，返回 True。如果我们正在寻找 `20` ， 现在哈希值为 `9` ，而槽 `9` 当前项为 `31` 。我们不能简单地返回 False，因为我们知道可能存在冲突。我们现在被迫做一个顺序搜索，从位置 `10` 开始寻找，直到我们找到项 `20` 或我们找到一个空槽。

线性探测的缺点是聚集的趋势;项在表中聚集。这意味着如果在相同的散列值处发生很多冲突，则将通过线性探测来填充多个周边槽。这将影响正在插入的其他项，正如我们尝试添加上面的项 `20` 时看到的。必须跳过一组值为 `0` 的值，最终找到开放位置。该聚集如 Figure 9 所示。

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 77 | 44 | 55 | 20 | 26 | 93 | 17 | None | None | 31 | 54 |

*Figure 9*

处理聚集的一种方式是扩展线性探测技术，使得不是顺序地查找下一个开放槽，而是跳过槽，从而更均匀地分布已经引起冲突的项。这将潜在地减少发生的聚集。 Figure 10 展示了使用 `加3` 探头进行碰撞识别时的项。 这意味着一旦发生碰撞，我们将查看第三个槽，直到找到一个空。

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 77 | 55 | None | 44 | 26 | 93 | 17 | 20 | None | 31 | 54 |

*Figure 10*

在冲突后寻找另一个槽的过程叫 `重新散列` 。使用简单的线性探测，rehash 函数是 `newhashvalue = rehash(oldhashvalue)` 其中 `rehash(pos)=(pos + 1)%sizeoftable` 。 `加3` rehash 可以定义为 `rehash(pos)=(pos + 3)%sizeoftable` 。一般来说， `rehash(pos)=(pos + skip)%sizeoftable` 。重要的是要注意，“跳过”的大小必须使得表中的所有槽最终都被访问。否则，表的一部分将不被使用。为了确保这一点，通常建议表大小是素数。这是我们在示例中使用 11 的原因。

线性探测思想的一个变种称为二次探测。代替使用常量 “跳过” 值，我们使用rehash 函数，将散列值递增 `1，3，5，7，9，` 依此类推。这意味着如果第一哈希值是 `h` ，则连续值是 `h + 1，h + 4，h + 9，h + 16` ，等等。换句话说，二次探测使用由连续完全正方形组成的跳跃。 Figure 11 展示了使用此技术放置之后的示例值。

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 77 | 44 | 20 | 55 | 26 | 93 | 17 | None | None | 31 | 54 |

*Figure 11*

用于处理冲突问题的替代方法是允许每个槽保持对项的集合（或链）的引用。链接允许许多项存在于哈希表中的相同位置。当发生冲突时，项仍然放在散列表的正确槽中。随着越来越多的项哈希到相同的位置，搜索集合中的项的难度增加。 Figure 12 展示了添加到使用链接解

决冲突的散列表时的项。

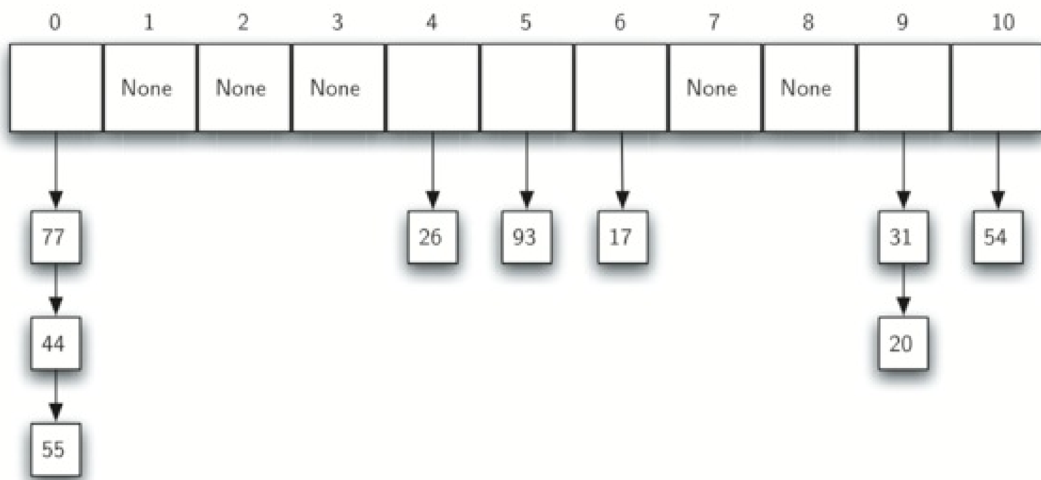



*Figure 12*

当我们要搜索一个项时，我们使用散列函数来生成它应该在的槽。由于每个槽都有一个集合，我们使用一种搜索技术来查找该项是否存在。优点是，平均来说，每个槽中可能有更少的项，因此搜索可能更有效。我们将在本节结尾处查看散列的分析。

## 5.5.3.实现 map 抽象数据类型

最有用的 Python 集合之一是字典。回想一下，字典是一种关联数据类型，你可以在其中存储键-值对。该键用于查找关联的值。我们经常将这个想法称为 `map` 。

map 抽象数据类型定义如下。该结构是键与值之间的关联的无序集合。map 中的键都是唯一的，因此键和值之间存在一对一的关系。操作如下。

- Map() 创建一个新的 map 。它返回一个空的 map 集合。
- put(key，val) 向 map 中添加一个新的键值对。如果键已经在 map 中，那么用新值替换旧值。
- get(key) 给定一个键，返回存储在 map 中的值或 None。
- del 使用 `del map[key]` 形式的语句从 map 中删除键值对。
- len() 返回存储在 map 中的键值对的数量。
- in 返回 True 对于 `key in map` 语句，如果给定的键在 map 中，否则为False。

字典一个很大的好处是，给定一个键，我们可以非常快速地查找相关的值。为了提供这种快速查找能力，我们需要一个支持高效搜索的实现。我们可以使用具有顺序或二分查找的列表，但是使用如上所述的哈希表将更好，因为查找哈希表中的项可以接近 O(1) 性能。

在 Listing 2 中，我们使用两个列表来创建一个实现 Map 抽象数据类型的HashTable 类。一个名为 `slots` 的列表将保存键项，一个称 `data` 的并行列表将保存数据值。当我们查找一个键时， `data` 列表中的相应位置将保存相关的数据值。我们将使用前面提出的想法将键列表视为哈希表。注意，哈希表的初始大小已经被选择为 11。尽管这是任意的，但是重要的是，大小是质数，使得冲突解决算法可以尽可能高效。

```
class HashTable:
    def __init__(self):
        self.size = 11
        self.slots = [None] * self.size
        self.data = [None] * self.size
```

*Listing 2*

hash 函数实现简单的余数方法。冲突解决技术是 `加1` rehash 函数的线性探测。 put 函数（见 Listing 3）假定最终将有一个空槽，除非 key 已经存在于 `self.slots` 中。 它计算原始哈希值，如果该槽不为空，则迭代 rehash 函数，直到出现空槽。如果非空槽已经包含 key，则旧数据值将替换为新数据值。

```
def put(self,key,data):
  hashvalue = self.hashfunction(key,len(self.slots))

  if self.slots[hashvalue] == None:
    self.slots[hashvalue] = key
    self.data[hashvalue] = data
  else:
    if self.slots[hashvalue] == key:
      self.data[hashvalue] = data  #replace
    else:
      nextslot = self.rehash(hashvalue,len(self.slots))
      while self.slots[nextslot] != None and \
                      self.slots[nextslot] != key:
        nextslot = self.rehash(nextslot,len(self.slots))

      if self.slots[nextslot] == None:
        self.slots[nextslot]=key
        self.data[nextslot]=data
      else:
        self.data[nextslot] = data #replace

def hashfunction(self,key,size):
     return key%size

def rehash(self,oldhash,size):
    return (oldhash+1)%size
```

*Listing 3*

同样，get 函数（见 Listing 4）从计算初始哈希值开始。如果值不在初始槽中，则 rehash 用于定位下一个可能的位置。注意，第 15 行保证搜索将通过检查以确保我们没有返回到初始槽来终止。如果发生这种情况，我们已用尽所有可能的槽，并且项不存在。

HashTable 类提供了附加的字典功能。我们重载 `__getitem__` 和 `__setitem__` 方法以允许使用 `[]` 访问。 这意味着一旦创建了HashTable，索引操作符将可用。

```
def get(self,key):
  startslot = self.hashfunction(key,len(self.slots))

  data = None
  stop = False
  found = False
  position = startslot
  while self.slots[position] != None and  \
                     not found and not stop:
     if self.slots[position] == key:
       found = True
       data = self.data[position]
     else:
       position=self.rehash(position,len(self.slots))
       if position == startslot:
           stop = True
  return data

def __getitem__(self,key):
    return self.get(key)

def __setitem__(self,key,data):
    self.put(key,data)
```

*Listing 4*

下面的会话展示了 HashTable 类的操作。首先，我们将创建一个哈希表并存储一些带有整数键和字符串数据值的项。

```
>>> H=HashTable()
>>> H[54]="cat"
>>> H[26]="dog"
>>> H[93]="lion"
>>> H[17]="tiger"
>>> H[77]="bird"
>>> H[31]="cow"
>>> H[44]="goat"
>>> H[55]="pig"
>>> H[20]="chicken"
>>> H.slots
[77, 44, 55, 20, 26, 93, 17, None, None, 31, 54]
>>> H.data
['bird', 'goat', 'pig', 'chicken', 'dog', 'lion',
       'tiger', None, None, 'cow', 'cat']
```

接下来，我们将访问和修改哈希表中的一些项。注意，正替换键 20 的值。

```
>>> H[20]
'chicken'
>>> H[17]
'tiger'
>>> H[20]='duck'
>>> H[20]
'duck'
>>> H.data
['bird', 'goat', 'pig', 'duck', 'dog', 'lion',
       'tiger', None, None, 'cow', 'cat']
>> print(H[99])
None
```

## 5.5.4.hash法分析

我们之前说过，在最好的情况下，散列将提供 O(1)，恒定时间搜索。然而，由于冲突，比较的数量通常不是那么简单。即使对散列的完整分析超出了本文的范围，我们可以陈述一些近似搜索项所需的比较数量的已知结果。

我们需要分析散列表的使用的最重要的信息是负载因子 λ。概念上，如果 λ 小，则碰撞的机会较低，这意味着项更可能在它们所属的槽中。如果 λ 大，意味着表正在填满，则存在越来越多的冲突。这意味着冲突解决更困难，需要更多的比较来找到一个空槽。使用链接，增加的碰撞意味着每个链上的项数量增加。

和以前一样，我们将有一个成功的搜索结果和不成功的。对于使用具有线性探测的开放寻址的成功搜索，平均比较数大约为 `1/2（1 + 1/(1-λ)）` ，不成功的搜索为 1/2(1+(1/1-λ)^2 ) 如果我们使用链接，则对于成功的情况，平均比较数目是 1+λ/2，如果搜索不成功，则简单地是 λ 比较次数。

# 5.6.排序

排序是以某种顺序从集合中放置元素的过程。例如，单词列表可以按字母顺序或按长度排序。城市列表可按人口，按地区或邮政编码排序。我们已经看到了许多能够从排序列表中获益的算法（回忆之前的回文例子和二分查找）。

有许多开发和分析的排序算法。表明排序是计算机科学的一个重要研究领域。对大量项进行排序可能需要大量的计算资源。与搜索一样，排序算法的效率与正在处理的项的数量有关。对于小集合，复杂的排序方法可能更麻烦，开销太高。另一方面，对于更大的集合，我们希望利用尽可能多的改进。在本节中，我们将讨论几种排序技术，并对它们的运行时间进行比较。

在分析特定算法之前，我们应该考虑可用于分析排序过程的操作。首先，必须比较两个值以查看哪个更小（或更大）。为了对集合进行排序，需要一些系统的方法来比较值，以查看是否有问题。比较的总数将是测量排序过程的最常见方法。第二，当值相对于彼此不在正确的位置时，可能需要交换它们。这种交换是一种昂贵的操作，并且交换的总数对于评估算法的整体效率也将是很重要的。

# 5.7.冒泡排序

冒泡排序需要多次遍历列表。它比较相邻的项并交换那些无序的项。每次遍历列表将下一个最大的值放在其正确的位置。实质上，每个项“冒泡”到它所属的位置。

Figure 1 展示了冒泡排序的第一次遍历。阴影项正在比较它们是否乱序。如果在列表中有 n 个项目，则第一遍有 n-1 个项需要比较。重要的是要注意，一旦列表中的最大值是一个对的一部分，它将不断地被移动，直到遍历完成。

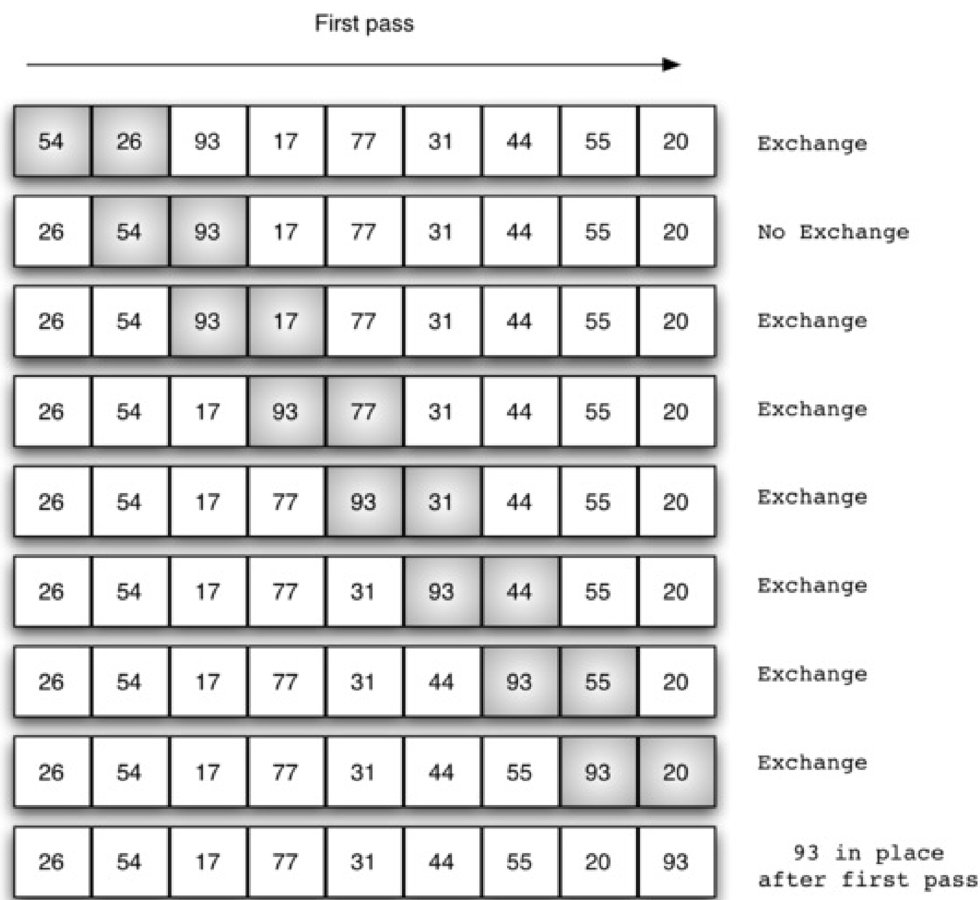



*Figure 1*

在第二次遍历的开始，现在最大的值已经在正确的位置。有 n-1 个项留下排序，意味着将有 n-2 对。由于每次通过将下一个最大值放置在适当位置，所需的遍历的总数将是 n-1。 在完成 n-1 遍之后，最小的项肯定在正确的位置，不需要进一步处理。 ActiveCode 1 显示完整的 `bubbleSort` 函数。它将列表作为参数，并根据需要交换项来修改它。

交换操作，有时称为 `swap`，在 Python 中与在大多数其他编程语言略有不同。通常，交换列表中的两个元素需要临时存储位置（额外的内存位置）。 代码片段如下

```
temp = alist[i]
alist[i] = alist[j]
alist[j] = temp
```

将交换列表中的第 i 项和第 j 项。没有临时存储，其中一个值将被覆盖。

在Python中，可以执行同时赋值。 语句 `a，b = b，a` 两个赋值语句同时完成（参见 Figure 2）。使用同时分配，交换操作可以在一个语句中完成。

ActiveCode 1 中的行 5-7 使用先前描述的三步过程执行 i 和第 i + 1 个项的交换。 注意，我们也可以使用同时分配来交换项目。

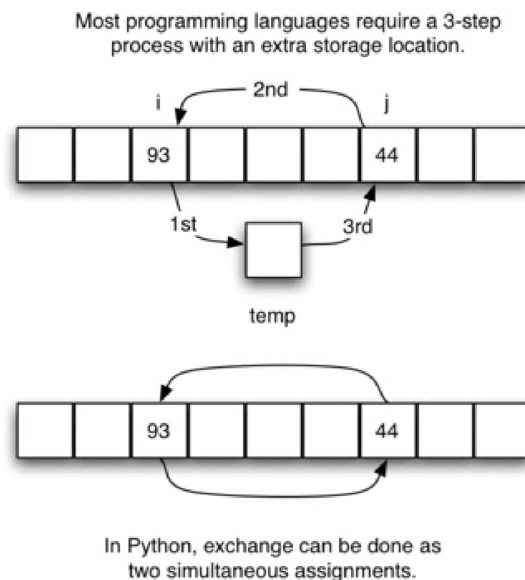



*Figure 2*

```
def bubbleSort(alist):
    for passnum in range(len(alist)-1,0,-1):
        for i in range(passnum):
            if alist[i]>alist[i+1]:
                temp = alist[i]
                alist[i] = alist[i+1]
                alist[i+1] = temp

alist = [54,26,93,17,77,31,44,55,20]
bubbleSort(alist)
print(alist)
```

*ActiveCode 1*

为了分析气泡排序，我们应该注意，不管项如何在初始列表中排列，将进行 n-1 次遍历以排序大小为 n 的列表。 Figure 1 展示了每次通过的比较次数。比较的总数是第 n-1 个整数的和。回想起来，前 n 个整数的和是 1/2n^2 + 1/2n。 第 n-1 个整数的和为 1/2n^2 + 1/2n -n，其为 1/2n^2 - 1/2n。 这仍然是 O(n^2 ) 比较。在最好的情况下，如果列表已经排序，则不会进行交换。 但是，在最坏的情况下，每次比较都会导致交换元素。 平均来说，我们交换了一半时间。

| Pass | Comparisons |
| --- | --- |
| 1 | $n-1$ |
| 2 | $n-2$ |
| 3 | $n-3$ |
| ... | ... |
| $n-1$ | $1$ |

*Table1*

冒泡排序通常被认为是最低效的排序方法，因为它必须在最终位置被知道之前交换项。 这些“浪费”的交换操作是非常昂贵的。 然而，因为冒泡排序遍历列表的整个未排序部分，它有能力做大多数排序算法不能做的事情。特别地，如果在遍历期间没有交换，则我们知道该列表已排序。 如果发现列表已排序，可以修改冒泡排序提前停止。这意味着对于只需要遍历几次列表，冒泡排序具有识别排序列表和停止的优点。 ActiveCode 2 展示了这种修改，通常称为 `短冒泡排序` 。

```
def shortBubbleSort(alist):
    exchanges = True
    passnum = len(alist)-1
    while passnum > 0 and exchanges:
       exchanges = False
       for i in range(passnum):
           if alist[i]>alist[i+1]:
               exchanges = True
               temp = alist[i]
               alist[i] = alist[i+1]
               alist[i+1] = temp
       passnum = passnum-1

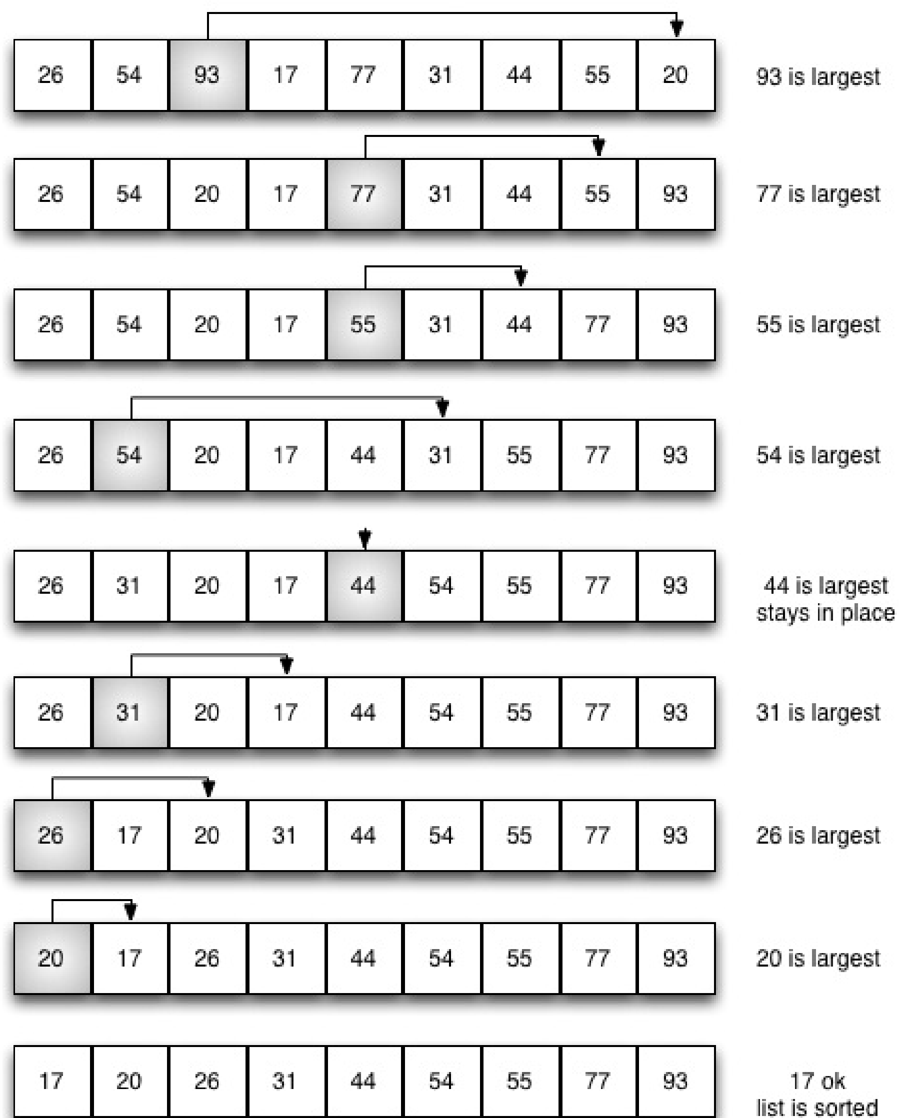
alist=[20,30,40,90,50,60,70,80,100,110]
shortBubbleSort(alist)
print(alist)
```

*Activecode 2*

# 5.8.选择排序

选择排序改进了冒泡排序，每次遍历列表只做一次交换。为了做到这一点，一个选择排序在他遍历时寻找最大的值，并在完成遍历后，将其放置在正确的位置。与冒泡排序一样，在第一次遍历后，最大的项在正确的地方。 第二遍后，下一个最大的就位。遍历 n-1 次排序 n 个项，因为最终项必须在第（n-1）次遍历之后。

Figure 3 展示了整个排序过程。在每次遍历时，选择最大的剩余项，然后放置在其适当位置。第一遍放置 93，第二遍放置 77，第三遍放置 55 等。 该函数展示在 ActiveCode 1 中。




*Activecode 1*

```
def selectionSort(alist):
   for fillslot in range(len(alist)-1,0,-1):
       positionOfMax=0
       for location in range(1,fillslot+1):
           if alist[location]>alist[positionOfMax]:
               positionOfMax = location

       temp = alist[fillslot]
       alist[fillslot] = alist[positionOfMax]
       alist[positionOfMax] = temp

alist = [54,26,93,17,77,31,44,55,20]
selectionSort(alist)
print(alist)
```

你可能会看到选择排序与冒泡排序有相同数量的比较，因此也是 O(n^2 )。 然而，由于交换数量的减少，选择排序通常在基准研究中执行得更快。 事实上，对于我们的列表，冒泡排序有 20 次交换，而选择排序只有 8 次。

# 5.9.插入排序

插入排序，尽管仍然是 O(n^2 )，工作方式略有不同。它始终在列表的较低位置维护一个排序的子列表。然后将每个新项 “插入” 回先前的子列表，使得排序的子列表称为较大的一个项。Figure 4 展示了插入排序过程。 阴影项表示算法进行每次遍历时的有序子列表。

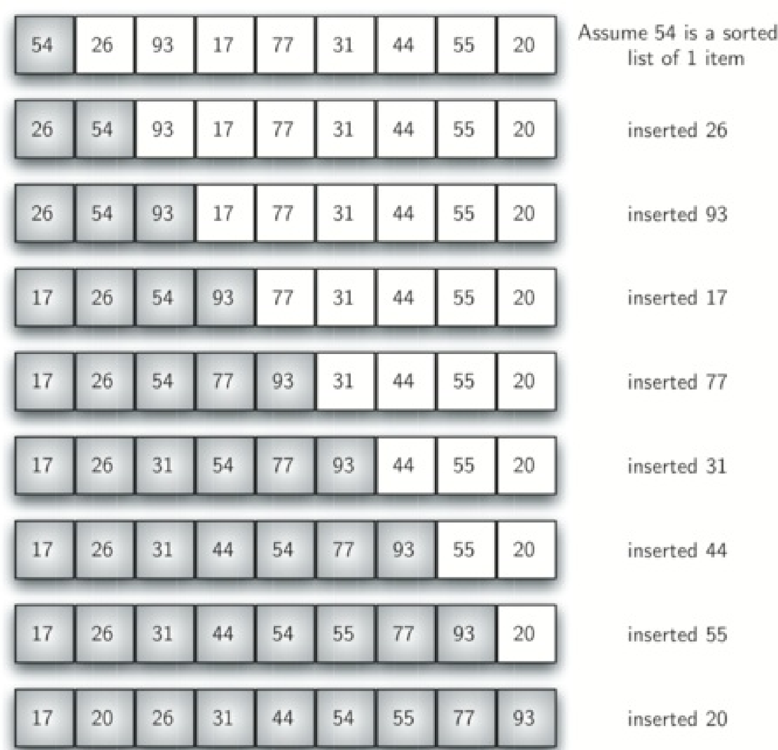



*Figure 4*

我们开始假设有一个项（位置 0 ）的列表已经被排序。在每次遍历时，对于每个项 1至 n-1，将针对已经排序的子列表中的项检查当前项。当我们回顾已经排序的子列表时，我们将那些更大的项移动到右边。 当我们到达较小的项或子列表的末尾时，可以插入当前项。

Figure 5 详细展示了第五次遍历。在该算法中的这一点，存在由 `17,26,54,77` 和 `93` 组成的五个项的排序子列表。我们插入 `31` 到已经排序的项。第一次与 93 比较导致 93 向右移位。77 和 54 也移位。 当遇到 26 时，移动过程停止，并且 31 被置于开放位置。现在我们有一个六个项的排序子列表。

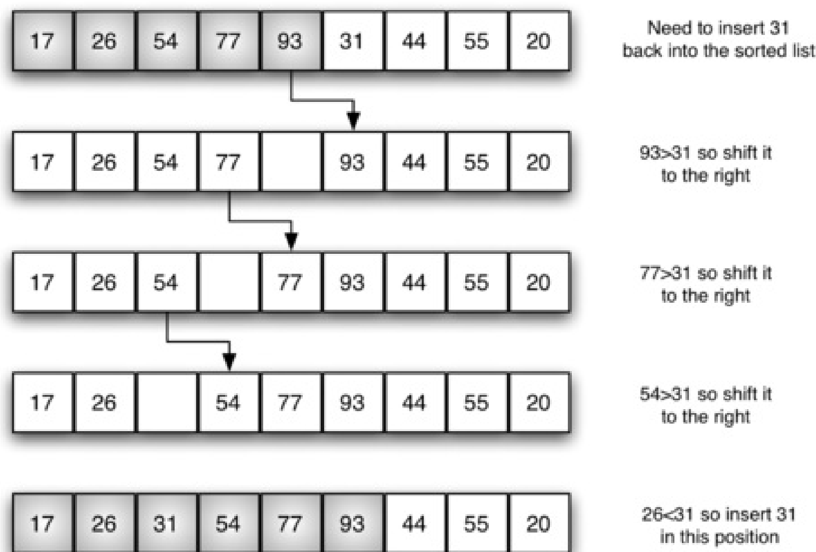



*Figure 5*

`insertSort` （ActiveCode 1）的实现展示了 存在 n-1 个遍历以对 n 个排序。从位置 1 开始迭代并移动位置到 n-1，因为这些是需要插回到排序子列表中的项。第 8 行执行移位操作，将值向上移动到列表中的一个位置，在其后插入。请记住，这不是像以前的算法中的完全交换。

插入排序的最大比较次数是 n-1 个整数的总和。同样，是 O(n^2 )。然而，在最好的情况下，每次通过只需要进行一次比较。这是已经排序的列表的情况。

关于移位和交换的一个注意事项也很重要。通常，移位操作只需要交换大约三分之一的处理工作，因为仅执行一次分配。在基准研究中，插入排序有非常好的性能。

```
def insertionSort(alist):
   for index in range(1,len(alist)):

     currentvalue = alist[index]
     position = index

     while position>0 and alist[position-1]>currentvalue:
         alist[position]=alist[position-1]
         position = position-1

     alist[position]=currentvalue

alist = [54,26,93,17,77,31,44,55,20]
insertionSort(alist)
print(alist)
```

*ActiveCode 1*

# 5.10.希尔排序

希尔排序（有时称为“递减递增排序”）通过将原始列表分解为多个较小的子列表来改进插入排序，每个子列表使用插入排序进行排序。 选择这些子列表的方式是希尔排序的关键。不是将列表拆分为连续项的子列表，希尔排序使用增量i（有时称为 `gap` ），通过选择 i 个项的所有项来创建子列表。

这可以在 Figure 6 中看到。该列表有九个项。如果我们使用三的增量，有三个子列表，每个子列表可以通过插入排序进行排序。完成这些排序后，我们得到如 Figure 7 所示的列表。虽然这个列表没有完全排序，但发生了很有趣的事情。 通过排序子列表，我们已将项目移动到更接近他们实际所属的位置。

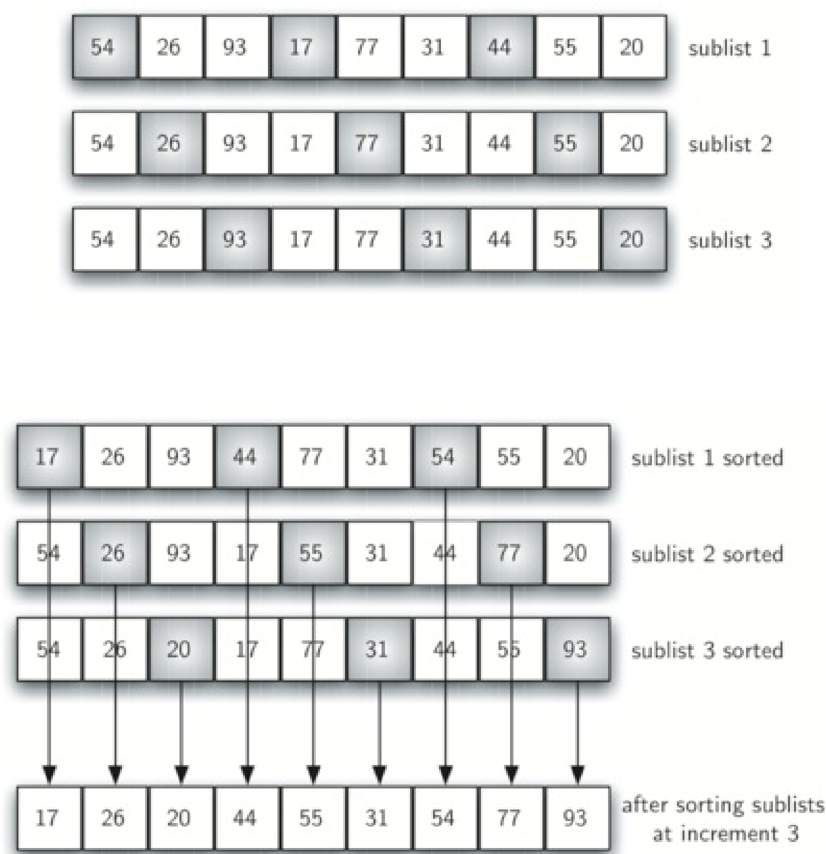



*Figure 6-7*

Figure 8 展示了使用增量为 1 的插入排序; 换句话说，标准插入排序。注意，通过执行之前的子列表排序，我们减少了将列表置于其最终顺序所需的移位操作的总数。对于这种情况，我们只需要四次移位完成该过程。

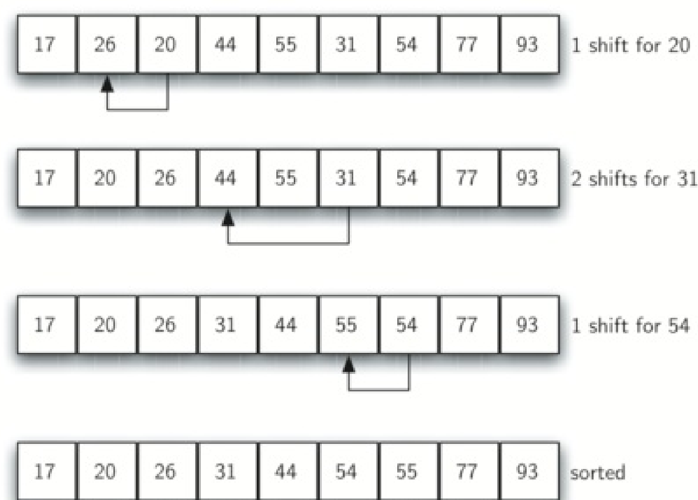



*Figure 8*

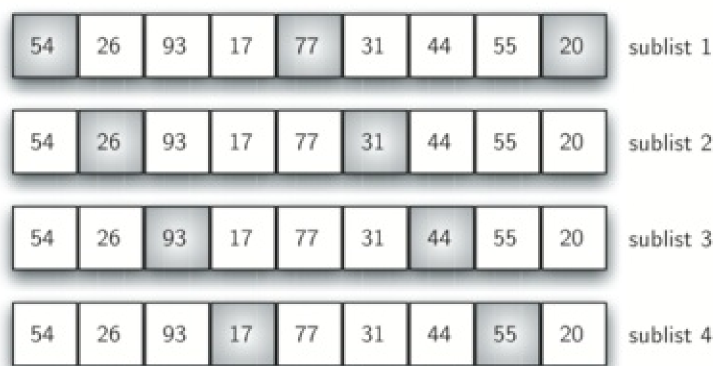



*Figure 9*

我们之前说过，增量的选择方式是希尔排序的独特特征。 ActiveCode 1中展示的函数使用不同的增量集。在这种情况下，我们从 n/2 子列表开始。下一次，n/4 子列表排序。 最后，单个列表按照基本插入排序进行排序。 Figure 9 展示了我们使用此增量的示例的第一个子列表。

```
def shellSort(alist):
    sublistcount = len(alist)//2
    while sublistcount > 0:

      for startposition in range(sublistcount):
        gapInsertionSort(alist,startposition,sublistcount)

      print("After increments of size",sublistcount,
                                   "The list is",alist)

      sublistcount = sublistcount // 2

def gapInsertionSort(alist,start,gap):
    for i in range(start+gap,len(alist),gap):

        currentvalue = alist[i]
        position = i

        while position>=gap and alist[position-gap]>currentvalue:
            alist[position]=alist[position-gap]
            position = position-gap

        alist[position]=currentvalue

alist = [54,26,93,17,77,31,44,55,20]
shellSort(alist)
print(alist)
```

*Activecode 1*

乍一看，你可能认为希尔排序不会比插入排序更好，因为它最后一步执行了完整的插入排序。 然而，结果是，该最终插入排序不需要进行非常多的比较（或移位），因为如上所述，该列表已经被较早的增量插入排序预排序。 换句话说，每个遍历产生比前一个“更有序”的列表。 这使得最终遍历非常有效。

虽然对希尔排序的一般分析远远超出了本文的范围，我们可以说，它倾向于落在 O(n) 和 O(n^2 ) 之间的某处，基于以上所描述的行为。对于 Listing 5中显示的增量，性能为 O(n^2 ) 。 通过改变增量，例如使用 `2^k -1（1,3,7,15,31等等）` ，希尔排序可以在 O（n^3/2 ）处执行。

# 5.11.归并排序

我们现在将注意力转向使用分而治之策略作为提高排序算法性能的一种方法。 我们将研究的第一个算法是归并排序。归并排序是一种递归算法，不断将列表拆分为一半。 如果列表为空或有一个项，则按定义（基本情况）进行排序。如果列表有多个项，我们分割列表，并递归调用两个半部分的合并排序。 一旦对这两半排序完成，就执行称为合并的基本操作。合并是获取两个较小的排序列表并将它们组合成单个排序的新列表的过程。 Figure 10 展示了我们熟悉的示例列表，它被mergeSort 分割。 Figure 11 展示了归并后的简单排序列表。

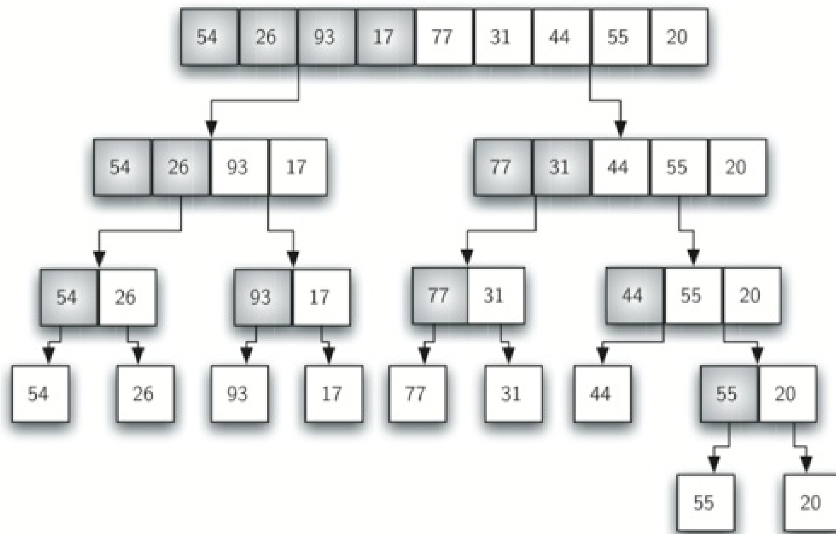


*Figure 10*

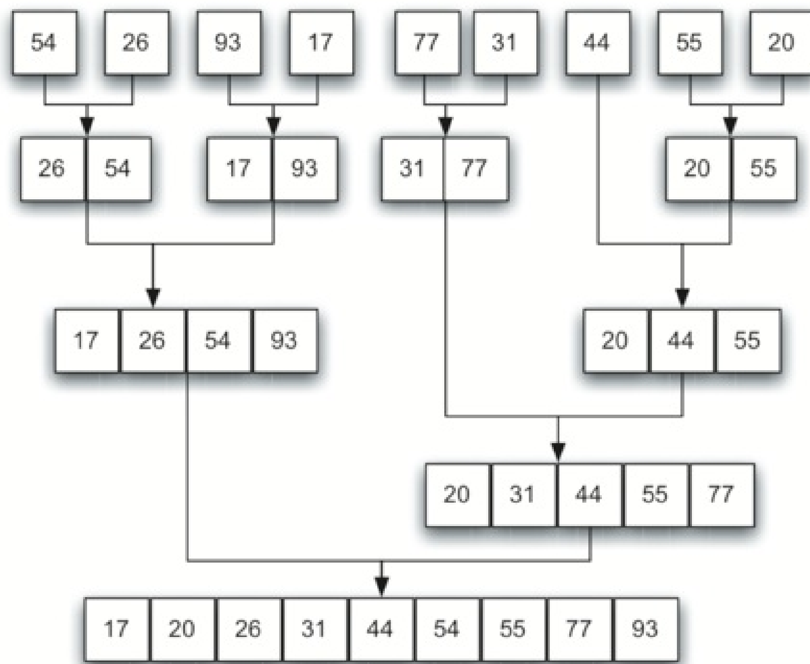



*Figure 11*

ActiveCode 1 中展示的 `mergeSort` 函数从询问基本情况开始。 如果列表的长度小于或等于 1，则我们已经有有序的列表，并且不需要更多的处理。另一方面，长度大于 1，那么我们使用 Python 切片操作来提取左右两半。 重要的是要注意，列表可能没有偶数个项。这并不重要，因为长度最多相差一个。

```
def mergeSort(alist):
    print("Splitting ",alist)
    if len(alist)>1:
        mid = len(alist)//2
        lefthalf = alist[:mid]
        righthalf = alist[mid:]

        mergeSort(lefthalf)
        mergeSort(righthalf)

        i=0
        j=0
        k=0
        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] < righthalf[j]:
                alist[k]=lefthalf[i]
                i=i+1
            else:
                alist[k]=righthalf[j]
                j=j+1
            k=k+1

        while i < len(lefthalf):
            alist[k]=lefthalf[i]
            i=i+1
            k=k+1

        while j < len(righthalf):
            alist[k]=righthalf[j]
            j=j+1
            k=k+1
    print("Merging ",alist)

alist = [54,26,93,17,77,31,44,55,20]
mergeSort(alist)
print(alist)
```

*Activecode1*

一旦在左半部分和右半部分（行8-9）上调用 mergeSort 函数，就假定它们已被排序。函数的其余部分（行11-31）负责将两个较小的排序列表合并成一个较大的排序列表。请注意，合并操作通过重复从排序列表中取最小的项目，将项目逐个放回原始列表（alist）。

mergeSort 函数已经增加了一个打印语句（行2），以显示在每次调用开始时排序的列表的内容。 还有一个打印语句（第32行）来显示合并过程。 脚本显示了在我们的示例列表中执行函数的结果。 请注意，`44,55` 和 `20` 的列表不会均匀分配。第一个分割出 [44]，第二个 [55,20]。 很容易看到分割过程最终产生可以立即与其他排序列表合并的列表。

为了分析 mergeSort 函数，我们需要考虑组成其实现的两个不同的过程。首先，列表被分成两半。我们已经计算过（在二分查找中）将列表划分为一半需要 log^n 次，其中 n 是列表的长度。第二个过程是合并。列表中的每个项将最终被处理并放置在排序的列表上。因此，大小为 n 的列表的合并操作需要 n 个操作。此分析的结果是 log^n 的拆分，其中每个操作花费 n，总共 nlog^n 。归并排序是一种 O(nlogn) 算法。

回想切片 是 O(k)，其中 k 是切片的大小。为了保证 mergeSort 是 O(nlog^n )，我们将需要删除 slice 运算符。这是可能的，如果当我们进行递归调用，我们简单地传递开始和结束索引与列表。我们把这作为一个练习。

重要的是注意，mergeSort 函数需要额外的空间来保存两个半部分，因为它们是使用切片操作提取的。如果列表很大，这个额外的空间可能是一个关键因素，并且在处理大型数据集时可能会导致此类问题。

# 5.12.快速排序

快速排序使用分而治之来获得与归并排序相同的优点，而不使用额外的存储。然而，作为权衡，有可能列表不能被分成两半。当这种情况发生时，我们将看到性能降低。

快速排序首先选择一个值，该值称为 `枢轴值` 。虽然有很多不同的方法来选择枢轴值，我们将使用列表中的第一项。枢轴值的作用是帮助拆分列表。枢轴值属于最终排序列表（通常称为拆分点）的实际位置，将用于将列表划分为快速排序的后续调用。

Figure 12 展示 54 将作为我们的第一个枢纽值。由于我们已经看过这个例子几次，我们知道 54 最终将会在当前持有 31 的位置。接下来将发生分区过程。它将找到拆分点，同时将其他项移动到列表的适当侧，小于或大于枢轴值。




*Figure 12*

分区从通过在列表中剩余项目的开始和结束处定位两个位置标记（我们称为左标记和右标记）开始（Figure 13中的位置 1 和 8 ）。分区的目标是移动相对于枢轴值位于错误侧的项，同时也收敛于分裂点。 Figure13展示了我们定位54的位置的过程。

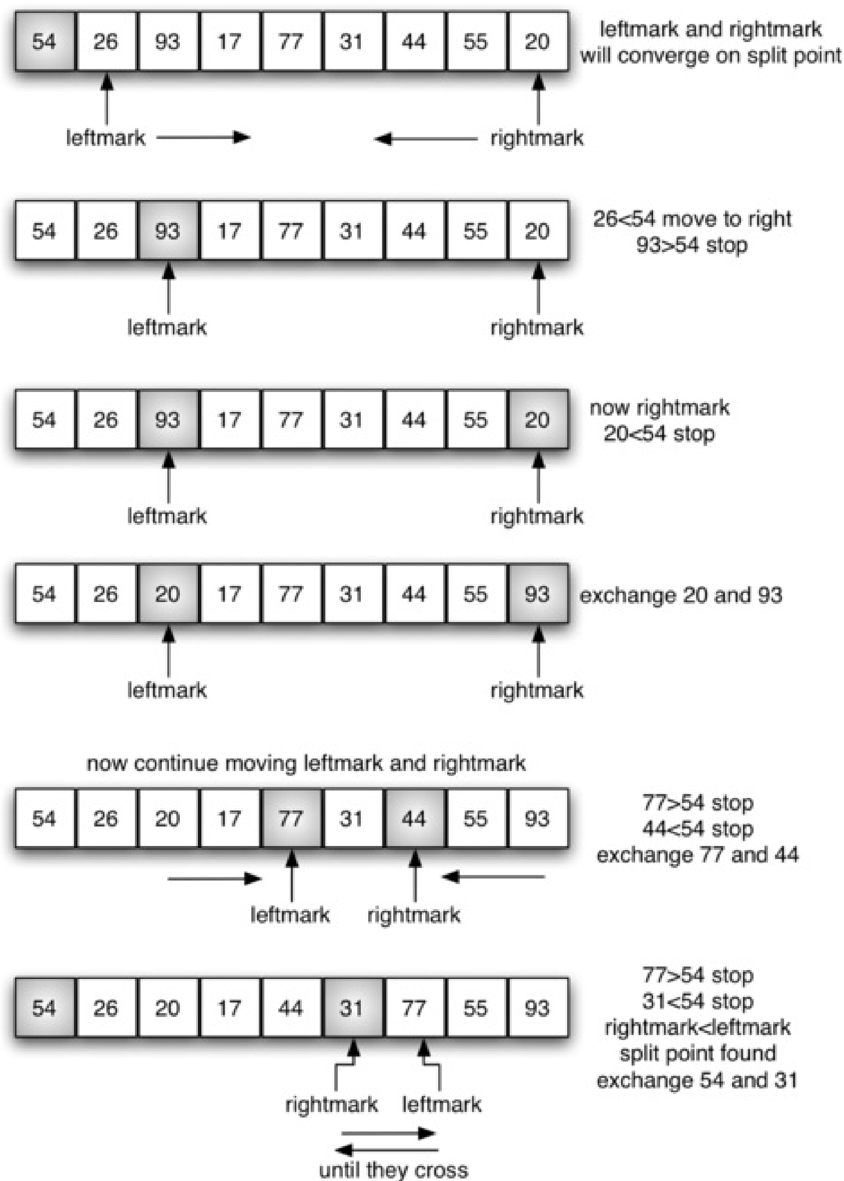



*Figure 13*

我们首先增加左标记，直到我们找到一个大于枢轴值的值。 然后我们递减右标，直到我们找到小于枢轴值的值。我们发现了两个相对于最终分裂点位置不适当的项。 对于我们的例子，这发生在 93 和 20。现在我们可以交换这两个项目，然后重复该过程。

在右标变得小于左标记的点，我们停止。右标记的位置现在是分割点。枢轴值可以与拆分点的内容交换，枢轴值现在就位（Figure 14）。 此外，分割点左侧的所有项都小于枢轴值，分割点右侧的所有项都大于枢轴值。现在可以在分割点处划分列表，并且可以在两半上递归调用快速排序。

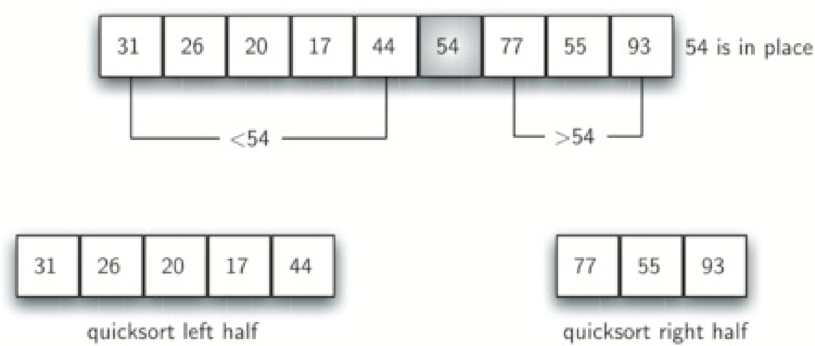



*Figure14*

ActiveCode 1中显示 `quickSort` 函数调用递归函数`quickSortHelper` 。 `quickSortHelper` 以与合并排序相同的基本情况开始。如果列表的长度小于或等于一，它已经排序。 如果它更大，那么它可以被分区和递归排序。 分区函数实现前面描述的过程。

```
def quickSort(alist):
   quickSortHelper(alist,0,len(alist)-1)

def quickSortHelper(alist,first,last):
   if first<last:

       splitpoint = partition(alist,first,last)

       quickSortHelper(alist,first,splitpoint-1)
       quickSortHelper(alist,splitpoint+1,last)


def partition(alist,first,last):
   pivotvalue = alist[first]

   leftmark = first+1
   rightmark = last

   done = False
   while not done:

       while leftmark <= rightmark and alist[leftmark] <= pivotvalue:
           leftmark = leftmark + 1

       while alist[rightmark] >= pivotvalue and rightmark >= leftmark:
           rightmark = rightmark -1

       if rightmark < leftmark:
           done = True
       else:
           temp = alist[leftmark]
           alist[leftmark] = alist[rightmark]
           alist[rightmark] = temp

   temp = alist[first]
   alist[first] = alist[rightmark]
   alist[rightmark] = temp


   return rightmark

alist = [54,26,93,17,77,31,44,55,20]
quickSort(alist)
print(alist)
```

要分析 `quickSort` 函数，请注意，对于长度为 n 的列表，如果分区总是出现在列表中间，则会再次出现 logn 分区。为了找到分割点，需要针对枢轴值检查 n 个项中的每一个。结果是 nlogn。此外，在归并排序过程中不需要额外的存储器。

不幸的是，在最坏的情况下，分裂点可能不在中间，并且可能非常偏向左边或右边，留下非常不均匀的分割。在这种情况下，对 n 个项的列表进行排序划分为对0 个项的列表和 n-1 个项目的列表进行排序。然后将 n-1 个划分的列表排序为大小为0的列表和大小为 n-2 的列表，以此类推。结果是具有递归所需的所有开销的 O(n) 排序。

我们之前提到过，有不同的方法来选择枢纽值。特别地，我们可以尝试通过使用称为中值三的技术来减轻一些不均匀分割的可能性。要选择枢轴值，我们将考虑列表中的第一个，中间和最后一个元素。在我们的示例中，这些是54,77和20.现在选择中值，在我们的示例中为54，并将其用于枢轴值（当然，这是我们最初使用的枢轴值）。想法是，在列表中的第一项不属于列表的中间的情况下，中值三将选择更好的“中间”值。当原始列表部分有序时，这将特别有用。我们将此枢轴值选择的实现作为练习。

# 5.13.总结

- 对于有序和无序列表，顺序搜索是 O(n)。
- 在最坏的情况下，有序列表的二分查找是 O(log^n )。
- 哈希表可以提供恒定时间搜索。
- 冒泡排序，选择排序和插入排序是 O(n^2 )算法。
- shell排序通过排序增量子列表来改进插入排序。它落在 O(n) 和 O(n^2 ) 之间。
- 归并排序是 O(nlog^n ），但是合并过程需要额外的空间。
- 快速排序是 O(nlog^n ），但如果分割点不在列表中间附近，可能会降级到O(n^2 ) 。它不需要额外的空间。

# 6.1. 目标

- 要理解树数据结构是什么，以及如何使用它。
- 查看树如何用于实现 map 数据结构。
- 使用列表实现树。
- 使用类和引用来实现树。
- 实现树作为递归数据结构。
- 使用堆实现优先级队列。

# 6.2.树的例子

现在我们已经研究了线性数据结构，如栈和队列，并且有一些递归的经验，我们将看一个称为树的常见数据结构。树在计算机科学的许多领域中使用，包括操作系统，图形，数据库系统和计算机网络。树数据结构与他们的植物表亲有许多共同之处。树数据结构具有根，分支和叶。自然界中的树和计算机科学中的树之间的区别在于树数据结构的根在顶部，其叶在底部。

在我们开始研究树形数据结构之前，让我们来看几个常见的例子。我们树的第一个例子是生物学的分类树。Figure 1 展示了一些动物的生物分类的实例。从这个简单的例子，我们可以了解树的几个属性。此示例演示的第一个属性是树是分层的。通过分层，我们的意思是树的层次结构，更接近顶部的是抽象的东西和底部附近是更具体的东西。层次结构的顶部是 `Kingdom` ，树的下一层（上面的层的“Children”）是 `Phylum` ，然后是 `Class` ，等等。然而，无论我们在分类树中有多深，所有的生物仍然是 `animals` 。

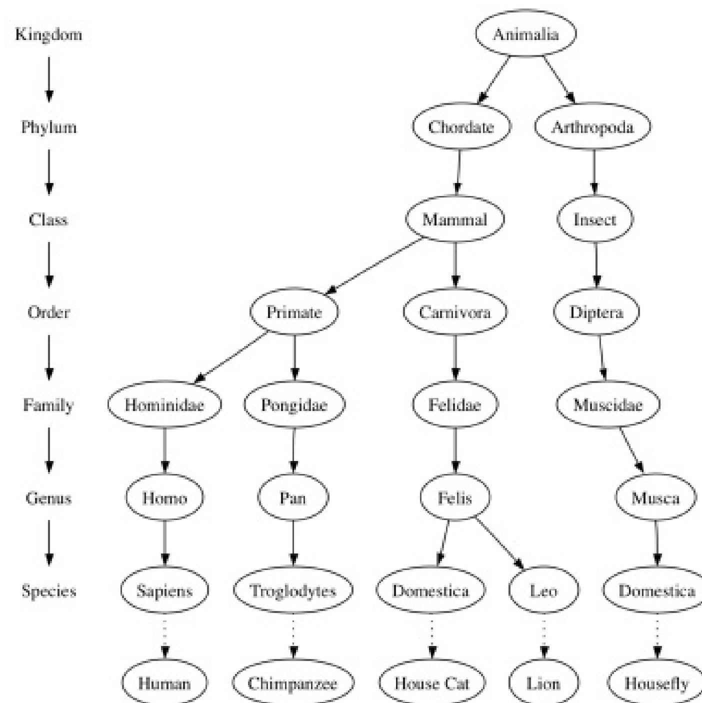



*Figure 1*

注意，你可以从树的顶部开始，并沿着圆圈和箭头一直到底部的路径。在树的每一层，我们可能问自己一个问题，然后遵循与我们的答案一致的路径。例如，我们可能会问，“这个动物是Chordate(脊椎动物)还是Arthropod(节肢动物)？”如果答案是“Chordate”，那么我们遵循这条路径，问“这个Chordate是 Mammal(哺乳动物)吗？”如果不是，我们就卡住了这个简化的例

子）。当我们在哺乳动物那层时，我们问“这个哺乳动物是Primate(灵长类动物)还是Carnivore(食肉动物)？”我们可以遵循以下路径，直到我们到达树的最底部，在那里我们有共同的名字。

树的第二个属性是一个节点的所有子节点独立于另一个节点的子节点。例如，Felis 有属于 Domestica 和 Leo 的孩子。Musca 也有一个名为 Domestica 的孩子，但它是一个不同的节点，并独立于 Felis 的 Domestica孩子。这意味着我们可以改变作为 Musca 的孩子的节点而不影响 Felis 的孩子。

第三个属性是每个叶节点是唯一的。我们可以指定从树的根到唯一地识别动物王国中的每个物种的叶的路径;例如，Animalia→→Chordate→→Mammal→→Carnivora→→Felidae→→Felis→→Domestica。

你可能每天使用的树结构的另一个示例是文件系统。在文件系统中，目录或文件夹被构造为树。Figure 2 说明了 Unix文件系统层次结构的一小部分。

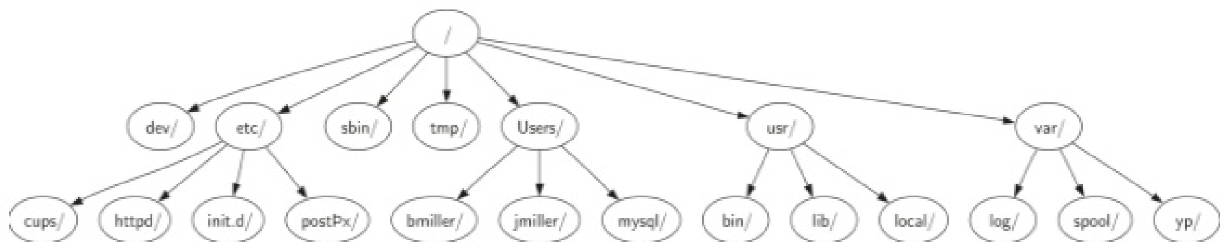



*Figure2*

文件系统树与生物分类树有很多共同之处。你可以遵循从根目录到任何目录的路径。 该路径将唯一标识该子目录（及其中的所有文件）。 树的另一个重要属性来源于它们的层次性质，你可以将树的整个部分（称为子树）移动到树中的不同位置，而不影响层次结构的较低级别。 例如，我们可以使用整个子树 /etc/，从根节点分离，并重新附加在 usr/ 下。 这将把 httpd 的唯一路径名从 /etc/httpd 更改为 /usr/etc/httpd，但不会影响 httpd 目录的内容或任何子级。

树的最后一个例子是网页。 以下是使用HTML编写的简单网页的示例。 Figure 3 展示了用于创建页面的每个 HTML 标记的树。

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
<head>
    <meta http-equiv="Content-Type"
          content="text/html; charset=utf-8" />
    <title>simple</title>
</head>
<body>
<h1>A simple web page</h1>
<ul>
    <li>List item one</li>
    <li>List item two</li>
</ul>
<h2><a href="http://www.cs.luther.edu">Luther CS </a><h2>
</body>
</html>
```

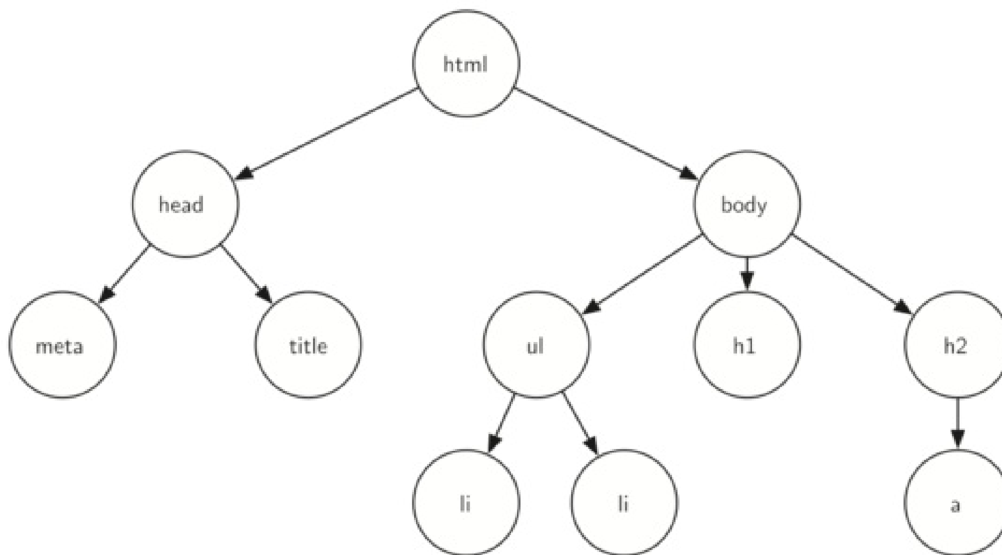



*Figure 3*

HTML源代码和伴随源的树说明了另一个层次结构。请注意，树的每个级别都对应于HTML标记内的嵌套级别。源中的第一个标记是 ，最后一个是</ html> 页面中的所有其余标记都是成对的。 如果你检查，你会看到这个嵌套属性在树的所有级别都是 true。

# 6.3.词汇和定义

我们已经看了树的示例，我们将正式定义树及其组件。

节点

节点是树的基本部分。它可以有一个名称，我们称之为“键”。节点也可以有附加信息。我们将这个附加信息称为“有效载荷”。虽然有效载荷信息不是许多树算法的核心，但在利用树的应用中通常是关键的。

边

边是树的另一个基本部分。边连接两个节点以显示它们之间存在关系。每个节点（除根之外）都恰好从另一个节点的传入连接。每个节点可以具有多个输出边。

根

树的根是树中唯一没有传入边的节点。在 Figure 2 中，/ 是树的根。

路径

路径是由边连接节点的有序列表。例如，
Mammal→→Carnivora→→Felidae→→Felis→→Domestica是一条路径。

子节点

具有来自相同传入边的节点 c 的集合称为该节点的子节点。在 Figure 2中，节点 log/，spool/ 和 yp/ 是节点 var/ 的子节点。

父节点

具有和它相同传入边的所连接的节点称为父节点。在 Figure 2 中，节点 var/ 是节点 log/，spool/ 和 yp/ 的父节点。

兄弟

树中作为同一父节点的子节点的节点被称为兄弟节点。节点 etc/ 和 usr/ 是文件系统树中的兄弟节点。

子树

子树是由父节点和该父节点的所有后代组成的一组节点和边。

叶节点

叶节点是没有子节点的节点。 例如，人类和黑猩猩是 Figure 1 中的叶节点。

层数

节点 n 的层数为从根结点到该结点所经过的分支数目。 例如，图1中的Felis节点的级别为五。根据定义，根节点的层数为零。

高度

树的高度等于树中任何节点的最大层数。 Figure 2 中的树的高度是 2。

现在已经定义了基本词汇，我们可以继续对树的正式定义。 事实上，我们将提供一个树的两个定义。 一个定义涉及节点和边。 第二个定义，将被证明是非常有用的，是一个递归定义。

定义一：树由一组节点和一组连接节点的边组成。树具有以下属性：

- 树的一个节点被指定为根节点。
- 除了根节点之外，每个节点 n 通过一个其他节点 p 的边连接，其中 p 是 n 的父节点。
- 从根路径遍历到每个节点路径唯一。
- 如果树中的每个节点最多有两个子节点，我们说该树是一个二叉树。

Figure 3 展示了适合定义一的树。边上的箭头指示连接的方向。

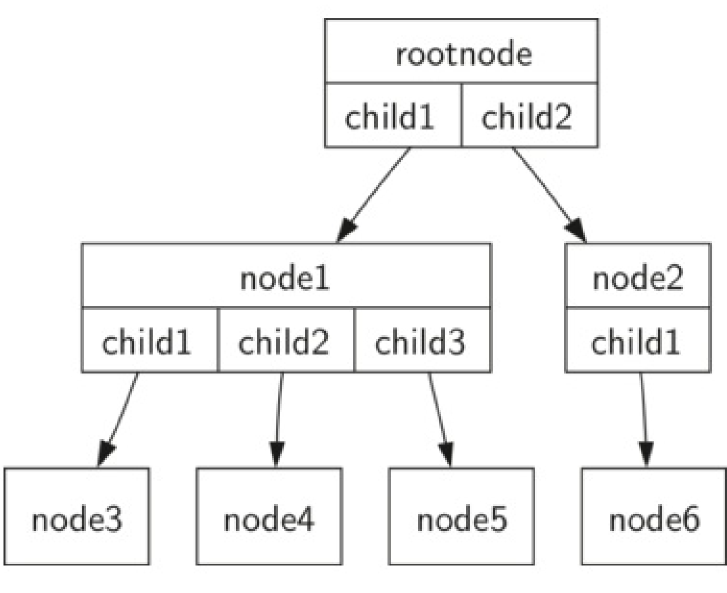



*Figure 3*

定义二：树是空的，或者由一个根节点和零个或多个子树组成，每个子树也是一棵树。每个子树的根节点通过边连接到父树的根节点。 Figure 4 说明了树的这种递归定义。使用树的递归定义，我们知道 Figure 4 中的树至少有四个节点，因为表示一个子树的每个三角形必须有一个根节点。 它可能有比这更多的节点，但我们不知道，除非我们更深入树。

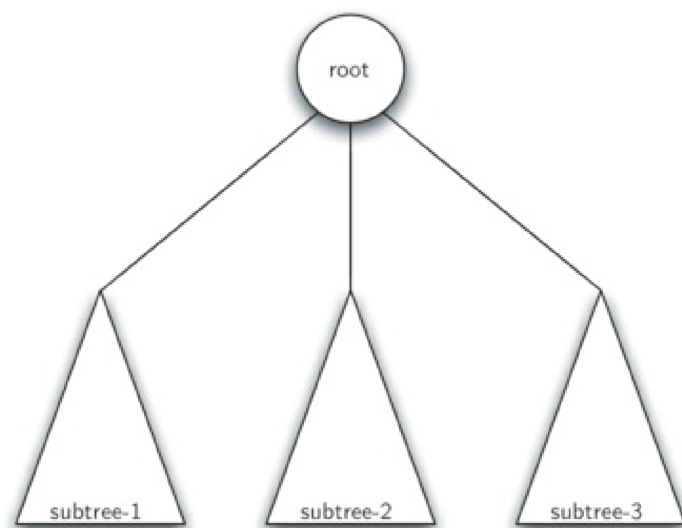



*Figure 4*

# 6.4.列表表示

在由列表表示的树中，我们将从 Python 的列表数据结构开始，并编写上面定义的函数。虽然将接口作为一组操作在列表上编写与我们实现的其他抽象数据类型有点不同，但这样做是有趣的，因为它为我们提供了一个简单的递归数据结构，我们可以直接查看和检查。在列表树的列表中，我们将根节点的值存储为列表的第一个元素。列表的第二个元素本身将是一个表示左子树的列表。列表的第三个元素将是表示右子树的另一个列表。为了说明这种存储技术，让我们看一个例子。 Figure 1 展示了一个简单的树和相应的列表实现。

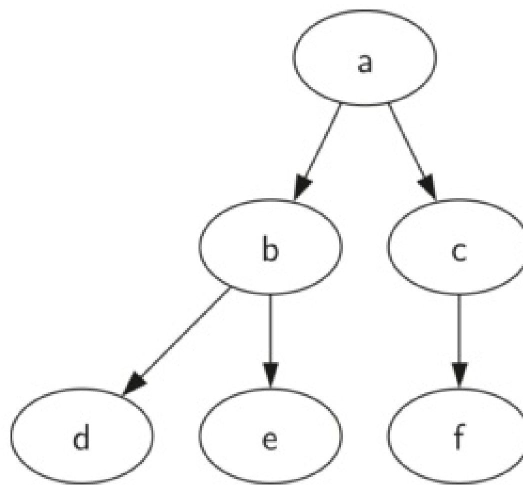



*Figure 1*

```
myTree = ['a',   #root
      ['b',  #left subtree
       ['d', [], []],
       ['e', [], []] ],
      ['c',  #right subtree
       ['f', [], []],
       [] ]
     ]
```

注意，我们可以使用标准列表索引来访问列表的子树。树的根是 `myTree[0]` ，根的左子树是 `myTree[1]` ，右子树是 `myTree[2]` 。 ActiveCode 1 说明了使用列表创建一个简单的树。一旦树被构建，我们可以访问根和左右子树。 该列表方法的一个非常好的属性是表示子树的列表的结构遵守树定义的结构; 结构本身是递归的！具有根值和两个空列表的子树是叶节点。列表方法的另一个很好的特性是它可以推广到一个有许多子树的树。在树超过二叉树的情况下，另一个子树只是另一个列表。

```
myTree = ['a', ['b', ['d',[],[]], ['e',[],[]] ], ['c', ['f',[],[]], []] ]
print(myTree)
print('left subtree = ', myTree[1])
print('root = ', myTree[0])
print('right subtree = ', myTree[2])
```

*Activecode 1*

让我们提供一些使我们能够使用列表作为树的函数来形式化树数据结构的这个定义。注意，我们不会定义一个二叉树类。我们写的函数只是帮助我们操纵一个标准列表，就像我们正在使用一棵树。

```
def BinaryTree(r):
    return [r, [], []]
```

`BinaryTree` 函数简单地构造一个具有根节点和两个子列表为空的列表。要将左子树添加到树的根，我们需要在根列表的第二个位置插入一个新的列表。我们必须小心。如果列表已经在第二个位置有东西，我们需要跟踪它，并沿着树向下把它作为我们添加的列表的左子节点。Listing 1 展示了插入左子节点的 Python 代码。

```
def insertLeft(root,newBranch):
    t = root.pop(1)
    if len(t) > 1:
        root.insert(1,[newBranch,t,[]])
    else:
        root.insert(1,[newBranch, [], []])
    return root
```

*Listing 1*

注意，要插入一个左子节点，我们首先获得与当前左子节点对应的（可能为空的）列表。然后我们添加新的左子树，添加旧的左子数作为新子节点的左子节点。这允许我们在任何位置将新节点拼接到树中。`insertRight` 的代码与 `insertLeft` 类似，如 Listing 2所示。

```
def insertRight(root,newBranch):
    t = root.pop(2)
    if len(t) > 1:
        root.insert(2,[newBranch,[],t])
    else:
        root.insert(2,[newBranch,[],[]])
    return root
```

*Listing 2*

为了完成这组树形函数（见 Listing 3），让我们编写一些访问函数来获取和设置根节点的值，以及获取左或右子树。

```
def getRootVal(root):
    return root[0]

def setRootVal(root,newVal):
    root[0] = newVal

def getLeftChild(root):
    return root[1]

def getRightChild(root):
    return root[2]
```

*Listing 3*

# 6.5.节点表示

我们的第二种表示树的方法使用节点和引用。在这种情况下，我们将定义一个具有根值属性的类，以及左和右子树。 由于这个表示更接近于面向对象的编程范例，我们将继续使用这个表示法用于本章的剩余部分。

使用节点和引用，我们认为树结构类似于 Figure 2 所示。

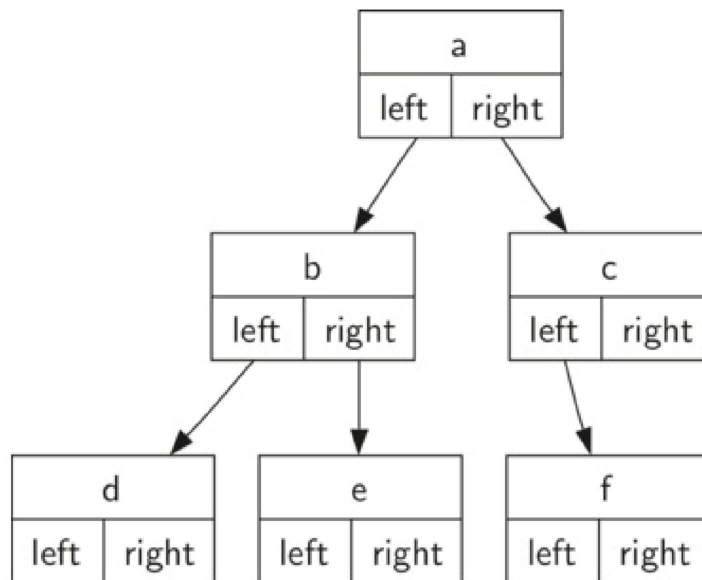



*Figure 2*

我们将从节点和引用方法的一个简单的类定义开始，如 Listing 4 所示。要记住这个表示重要的事情是 `left` 和 `right` 的属性将成为对 `BinaryTree` 类的其他实例的引用。 例如，当我们在树中插入一个新的左子节点时，我们创建另一个 `BinaryTree` 实例，并在根节点中修改 `self.leftChild` 来引用新树节点。

```
class BinaryTree:
    def __init__(self,rootObj):
        self.key = rootObj
        self.leftChild = None
        self.rightChild = None
```

*Listing 4*

请注意，在 Listing 4 中，构造函数希望获取某种对象存储在根中。 就像你可以在列表中存储任何你喜欢的对象一样，树的根对象可以是对任何对象的引用。 对于我们的先前示例，我们将存储节点的名称作为根值。使用节点和引用来表示 Figure 2 中的树，我们将创建 `BinaryTree` 类的六个实例。

接下来，我们来看看需要构建超出根节点的树的函数。要向树中添加一个左子树，我们将创建一个新的二叉树对象，并设置根的左边属性来引用这个新对象。 `insertLeft` 的代码如 Listing 5所示。

```
def insertLeft(self,newNode):
    if self.leftChild == None:
        self.leftChild = BinaryTree(newNode)
    else:
        t = BinaryTree(newNode)
        t.leftChild = self.leftChild
        self.leftChild = t
```

*Listing 5*

我们必须考虑两种插入情况。 第一种情况的特征没有现有左孩子的节点。当没有左孩子时，只需向树中添加一个节点。 第二种情况的特征在于具有现有左孩子的节点。在第二种情况下，我们插入一个节点并将现有的子节点放到树中的下一个层。第二种情况由 Listing 5 第 4 行的 else 语句处理。

`insertRight` 的代码必须考虑一组对称的情况。没有右孩子，或者我们在根和现有右孩子之间插入节点。 插入代码如 Listing 6 所示。

```
def insertRight(self,newNode):
    if self.rightChild == None:
        self.rightChild = BinaryTree(newNode)
    else:
        t = BinaryTree(newNode)
        t.rightChild = self.rightChild
        self.rightChild = t
```

*Listing 6*

为了完成一个简单二叉树数据结构的定义，我们将实现获取 左和右孩子（见 Listing 7 ）以及根值的方法。

```
def getRightChild(self):
    return self.rightChild

def getLeftChild(self):
    return self.leftChild

def setRootVal(self,obj):
    self.key = obj

def getRootVal(self):
    return self.key
```

*Listing 7*

现在我们有了创建和操作二叉树的所有部分，让我们使用它们来检查结构。 我们使用节点 a 作为根的简单树，并将节点 b 和 c 添加为子节点。 ActiveCode 1 创建树并查看存储在 key，left 和 right 中的一些值。注意，根的左和右孩子本身是 BinaryTree 类的不同实例。正如我们在对树的原始递归定义中所说的，这允许我们将二叉树的任何子项视为二叉树本身。

```
class BinaryTree:
    def __init__(self,rootObj):
        self.key = rootObj
        self.leftChild = None
        self.rightChild = None

    def insertLeft(self,newNode):
        if self.leftChild == None:
            self.leftChild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.leftChild = self.leftChild
            self.leftChild = t

    def insertRight(self,newNode):
        if self.rightChild == None:
            self.rightChild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.rightChild = self.rightChild
            self.rightChild = t


    def getRightChild(self):
        return self.rightChild

    def getLeftChild(self):
        return self.leftChild

    def setRootVal(self,obj):
        self.key = obj

    def getRootVal(self):
        return self.key


r = BinaryTree('a')
print(r.getRootVal())
print(r.getLeftChild())
r.insertLeft('b')
print(r.getLeftChild())
print(r.getLeftChild().getRootVal())
r.insertRight('c')
print(r.getRightChild())
print(r.getRightChild().getRootVal())
r.getRightChild().setRootVal('hello')
print(r.getRightChild().getRootVal())
```

# 6.6.分析树

随着我们的树数据结构的实现完成，我们现在看一个例子，说明如何使用树来解决一些真正的问题。在本节中，我们将讨论分析树。 分析树可以用于表示诸如句子或数学表达式的真实世界构造。

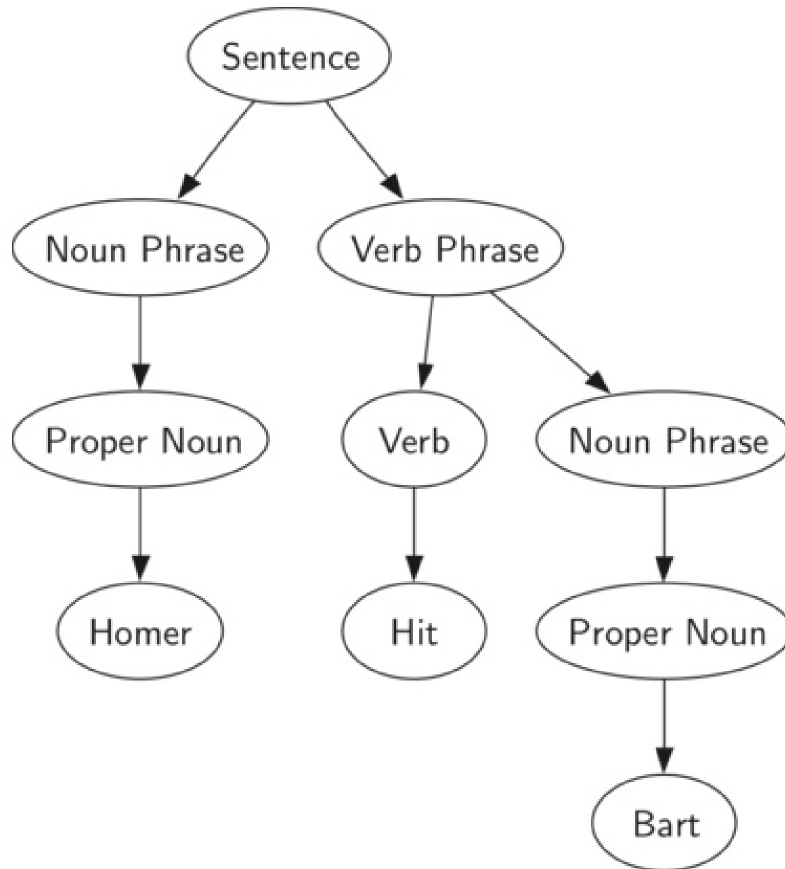



*Figure 1*

Figure 1 展示了一个简单句子的层次结构。 将句子表示为树结构允许我们通过使用子树来处理句子的各个部分。

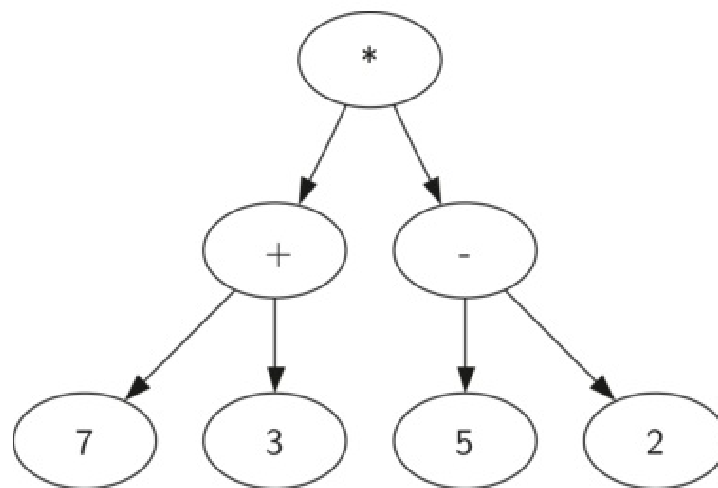



*Figure 2*

我们还可以表示诸如 `((7 + 3)*(5-2))` 数学表达式作为分析树，如 Figure 2 所示。我们早看过完全括号表达式，所以我们知道这个表达式是什么？我们知道乘法具有比加法或减法更高的优先级。由于括号，我们知道在做乘法之前，我们必须计算括号里面的加法和减法表达式。树的层次结构有助于我们了解整个表达式的求值顺序。在我们计算顶层乘法之前，我们必须计算子树中的加法和减法。作为左子树的加法结果为10。减法，即右子树，计算结果为3。使用树的层次结构，我们可以简单地用一个节点替换整个子树，一旦我们计算了表达式中这些子树。这个替换过程给出了 Figure 3 所示的简化树。

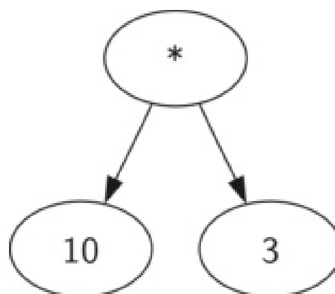



*Figure 3*

在本节的其余部分，我们将更详细地检查分析树。 特别的，我们会看

- 如何从完全括号的数学表达式构建分析树。
- 如何评估存储在分析树中的表达式。
- 如何从分析树中恢复原始数学表达式。

构建分析树的第一步是将表达式字符串拆分成符号列表。 有四种不同的符号要考虑：左括号，右括号，运算符和操作数。 我们知道，每当我们读一个左括号，我们开始一个新的表达式，因此我们应该创建一个新的树来对应于该表达式。 相反，每当我们读一个右括号，我们

就完成了一个表达式。 我们还知道操作数将是叶节点和它们的操作符的子节点。 最后，我们知道每个操作符都将有一个左和右孩子。

使用上面的信息，我们可以定义四个规则如下：

- 如果当前符号是 `'('` ，添加一个新节点作为当前节点的左子节点，并下降到左子节点。
- 如果当前符号在列表 `['+'，' - '，'/'，'*']` 中，请将当前节点的根值设置为由当前符号表示的运算符。 添加一个新节点作为当前节点的右子节点，并下降到右子节点。
- 如果当前符号是数字，请将当前节点的根值设置为该数字并返回到父节点。
- 如果当前令牌是 `')'` ，则转到当前节点的父节点。

在编写 Python 代码之前，让我们看看上面列出的规则的一个例子。我们将使用表达式 `（3+（4 * 5））` 。 我们将把这个表达式解析成下面的字符标记列表 `['('，'3'，'+'，'('，'4'，'*'，'5'，')'，')']` 。 最初，我们将使用由空根节点组成的分析树开始。 Figure 4 展示了当每个新符号被处理时分析树的结构和内容。


3
3
+
3
+
3
+
4

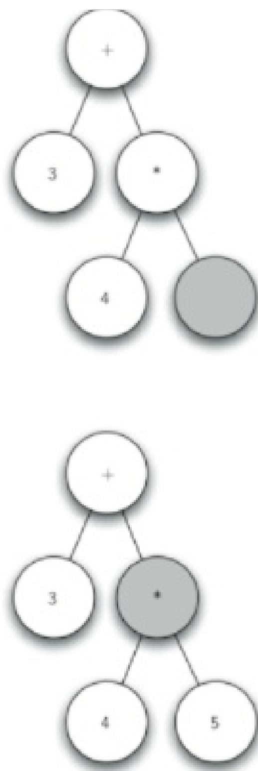


*Figure 4*

使用 Figure 4，让我们一步一步地浏览示例：

a. 创建一个空树。

b. 读取 ( 作为第一个标记。按规则1，创建一个新节点作为根的左子节点。使当前节点到这个新子节点。

c. 读取 3 作为下一个符号。按照规则3，将当前节点的根值设置为3，使当前节点返回到父节点。

d. 读取 + 作为下一个符号。根据规则2，将当前节点的根值设置为+，并添加一个新节点作为右子节点。新的右子节点成为当前节点。

e. 读取 ( 作为下一个符号，按规则1，创建一个新节点作为当前节点的左子节点，新的左子节点成为当前节点。

f. 读取 4 作为下一个符号。根据规则3，将当前节点的值设置为 4。使当前节点返回到父节点。

g. 读取 作为下一个符号。根据规则2，将当前节点的根值设置为 ，并创建一个新的右子节点。新的右子节点成为当前节点。

h. 读取 5 作为下一个符号。根据规则3，将当前节点的根值设置为5。使当前节点返回到父节点。

i. 读取 ) 作为下一个符号。根据规则4，使当前节点返回到父节点。

j. 读取 ) 作为下一个符号。根据规则4，使当前节点返回到父节点 + 。没有+ 的父节点，所以我们完成创建。

从上面的例子，很明显，我们需要跟踪当前节点以及当前节点的父节点。树接口为我们提供了一种通过 `getLeftChild` 和 `getRightChild` 方法获取节点的子节点的方法，但是我们如何跟踪父节点呢？当我们遍历树时，保持跟踪父对象的简单解决方案是使用栈。每当我们想下降到当前节点的子节点时，我们首先将当前节点入到栈上。当我们想要返回到当前节点的父节点时，我们将父节点从栈中弹出。

使用上述规则，以及 `Stack` 和 `BinaryTree` 操作，我们现在可以编写一个Python 函数来创建一个分析树。我们的分析树生成器的代码见 ActiveCode 1。

```
from pythonds.basic.stack import Stack
from pythonds.trees.binaryTree import BinaryTree

def buildParseTree(fpexp):
    fplist = fpexp.split()
    pStack = Stack()
    eTree = BinaryTree('')
    pStack.push(eTree)
    currentTree = eTree
    for i in fplist:
        if i == '(':
            currentTree.insertLeft('')
            pStack.push(currentTree)
            currentTree = currentTree.getLeftChild()
        elif i not in ['+', '-', '*', '/', ')']:
            currentTree.setRootVal(int(i))
            parent = pStack.pop()
            currentTree = parent
        elif i in ['+', '-', '*', '/']:
            currentTree.setRootVal(i)
            currentTree.insertRight('')
            pStack.push(currentTree)
            currentTree = currentTree.getRightChild()
        elif i == ')':
            currentTree = pStack.pop()
        else:
            raise ValueError
    return eTree

pt = buildParseTree("( ( 10 + 5 ) * 3 )")
pt.postorder()  #defined and explained in the next section
```

*Activecode1*

用于构建分析树的四个规则被编码为 ActiveCode 1 的行 11,15,19 和 24 上的 if 语句的前四个子句。在每种情况下，可以看到代码实现了如上所述的规则，与几个调用 BinaryTree 或 Stack 方法。我们在这个函数中唯一的错误检查是在 else 子句中，如果我们从列表中得到一个我们不认识的 token，我们引发一个ValueError异常。

现在我们已经构建了一个分析树，我们可以用它做什么？作为第一个例子，我们将编写一个函数来评估分析树，返回数值结果。要写这个函数，我们将利用树的层次性。回想一下 Figure 2。我们可以用 Figure 3 中所示的简化树替换原始树。这表明我们可以编写一个算法，通过递归地评估每个子树来评估一个分析树。

正如我们对过去的递归算法所做的，我们将通过识别基本情况来开始递归评价函数的设计。对树进行操作的递归算法的基本情况是检查叶节点。在分析树中，叶节点将始终是操作数。由于整数和浮点等数值对象不需要进一步解释，因此 `evaluate` 函数可以简单地返回存储在叶节点中的值。将函数移向基本情况的递归步骤是在当前节点的左子节点和右子节点上调用 evaluate。递归调用有效地使我们沿着树向着叶节点移动。

为了将两个递归调用的结果放在一起，我们可以简单地将存储在父节点中的运算符应用于从评估这两个子节点返回的结果。在 Figure 3的示例中，我们看到根的两个孩子计算得出结果，即 10 和 3。应用乘法运算符给我们一个最终结果 30。

递归求值函数的代码如 Listing 1 所示。首先，我们获取对当前节点的左子节点和右子节点的引用。如果左和右孩子都为 None ，那么我们知道当前节点实际上是一个叶节点。此检查在行 7。如果当前节点不是叶节点，请查找当前节点中的运算符，并将其应用于递归计算左右子节点的结果。

为了实现算术，我们使用具有键 `'+'，' - '，'*'` 和 `'/'` 的字典。存储在字典中的值是来自 Python 的运算符模块的函数。运算符模块为我们提供了许多常用操作符的功能。当我们在字典中查找一个运算符时，检索相应的函数对象。由于检索的对象是一个函数，我们可以用通常的方式 `function(param1，param2)` 调用它。因此，查找 `opers['+'](2,2)` 等效于 `operator.add(2,2)` 。

```
def evaluate(parseTree):
    opers = {'+':operator.add, '-':operator.sub, '*':operator.mul, '/':operator.truedi
v}

    leftC = parseTree.getLeftChild()
    rightC = parseTree.getRightChild()

    if leftC and rightC:
        fn = opers[parseTree.getRootVal()]
        return fn(evaluate(leftC),evaluate(rightC))
    else:
        return parseTree.getRootVal()
```

*Listing 1*

最后，我们将跟踪我们在 Figure 4 中创建的分析树上的求值函数。当我们首先调用 evaluate 时，我们将整个树的根作为参数 parseTree 传递。然后我们获得对左和右孩子的引用，以确保它们存在。递归调用发生在第 9 行。我们首先在树的根中查找运算符，它是 `'+'` 。 `'+'` 操作符映射到 `operator.add` 函数调用，它接受两个参数。像 Python 函数调用一样，Python 做的第一件事是计算传递给函数的参数。在这种情况下，两个参数都是对我们的 evaluate 函数的递归函数调用。使用从左到右的计算，第一个递归调用向左。在第一个递归调用中，赋值函数给出左子树。我们发现节点没有左或右孩子，所以我们得到一个叶节点。当我们在叶节点时，我们只是返回存储在叶节点中的值作为计算的结果。在这种情况下，我们返回整数 3 。

在这一点上，我们有一个参数对 `operator.add` 的顶层调用进行求值。但我们还没有完成。继续从左到右的参数计算，我们现在进行递归调用来评估根的右孩子。我们发现节点有一个左和右孩子，所以我们查找存储在这个节点 '' 运算符，并使用左和右孩子作为参数调用此函数。你可以看到，两个递归调用都到了叶节点，分别计算结果为整数 *4* 和 *5*。使用两个参数求值，我们返回 *`operator.mul(4,5)`* 的结果。在这一点上，我们已经计算了顶级 *`“+”`* 运算符的操作数，剩下的所有操作都完成对 *`operator.add(3,20)`* 的调用。对于`（*3*+（*4* 5））`的整个表达式树的计算结果是 23。

# 6.7.树的遍历

我们已经见到了树数据结构的基本功能，现在是看树的一些额外使用模式的时候了。这些使用模式可以分为我们访问树节点的三种方式。有三种常用的模式来访问树中的所有节点。这些模式之间的差异是每个节点被访问的顺序。我们称这种访问节点方式为“遍历”。我们将看到三种遍历方式称为 `前序，中序` 和 `后序` 。让我们更仔细地定义这三种遍历方式，然后看看这些模式有用的一些例子。

前序 在前序遍历中，我们首先访问根节点，然后递归地做左侧子树的前序遍历，随后是右侧子树的递归前序遍历。 中序 在一个中序遍历中，我们递归地对左子树进行一次遍历，访问根节点，最后递归遍历右子树。 后序 在后序遍历中，我们递归地对左子树和右子树进行后序遍历，然后访问根节点。

让我们看一些例子，来说明这三种遍历。首先看前序遍历。作为遍历的树的示例，我们将把这本书表示为树。这本书是树的根，每一章都是根节点的一个孩子。章节中的每个章节都是章节的子节点，每个小节都是章节的子节点，依此类推。Figure 5 展示了一本只有两章的书的有限版本。注意，遍历算法适用于具有任意数量子节点的树，但是我们现在使用二叉树。

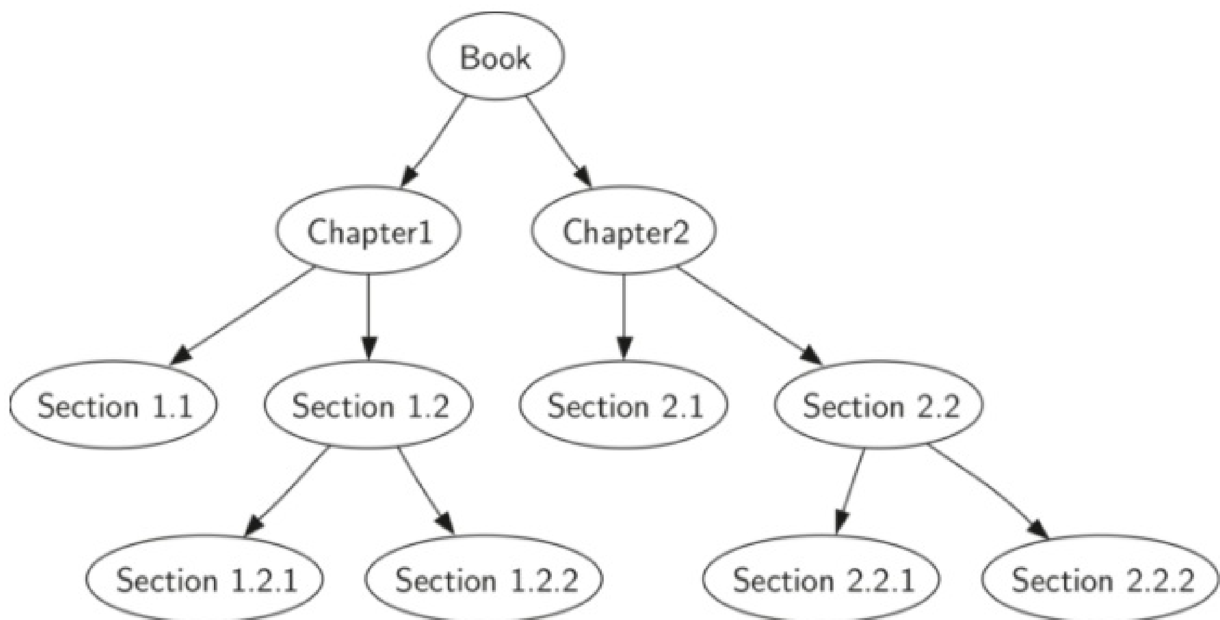



*Figure 5*

假设你想从前到后读这本书。前序遍历给你正确的顺序。从树的根（Book节点）开始，我们将遵循前序遍历指令。我们递归调用左孩子的 `preorder` ，在这种情况下是 `Chapter1` 。我们再次递归调用左孩子的 `preorder` 来得到 `Section 1.1` 。由于 `Section 1.1` 没有子节点，我们不再进行任何额外的递归调用。当我们完成 `Section 1.1` ，我们将树向上移动到 `Chapter1` 。此时，我们仍然需要访问 `Chapter1` 的右子树 `Section 1.2` 。和前面一样，我

们访问左子树，它将我们带到 `Section 1.2.1` ，然后访问 `Section 1.2.2` 。在 `Section 1.2` 完成后，我们返回到 `Chapter1` 。然后，我们返回到 `Book` 节点，并按照相同过程遍历 `Chapter2` 。

编写树遍历的代码惊人地优雅，主要是因为遍历是递归写的。Listing 2 展示了用于二叉树的前序遍历的 Python 代码。

你可能想知道，编写像前序遍历算法的最好方法是什么？是一个简单地使用树作为数据结构的函数，还是树数据结构本身的方法？Listing 2 展示了作为外部函数编写的前序遍历的版本，它将二叉树作为参数。外部函数特别优雅，因为我们的基本情况只是检查树是否存在。如果树参数为 None，那么函数返回而不采取任何操作。

```
def preorder(tree):
    if tree:
        print(tree.getRootVal())
        preorder(tree.getLeftChild())
        preorder(tree.getRightChild())
```

*Listing 2*

我们也可以实现 `preorder` 作为 `BinaryTree` 类的方法。Listing 3 中展示了将 `preorder` 实现为内部方法的代码。注意当我们将代码从内部移动到外部时会发生什么。 一般来说，我们只需用 `self` 替换 `tree` 。 但是，我们还需要修改基本情况。内部方法必须在进行前序的递归调用之前检查左和右孩子的存在。

```
def preorder(self):
    print(self.key)
    if self.leftChild:
        self.leftChild.preorder()
    if self.rightChild:
        self.rightChild.preorder()
```

*Listing 3*

以上哪种方式实现前序最好？ 答案是在这种情况下，实现前序作为外部函数可能更好。原因是你很少只是想遍历树。在大多数情况下，将要使用其中一个基本的遍历模式来完成其他任务。 事实上，我们将在下面的例子中看到后序遍历模式与我们前面编写的用于计算分析树的代码非常接近。 因此，我们用外部函数实现其余的遍历。

Listing 4 中所示的后序遍历算法几乎与前序遍历顺序相同，只是将 print 调用移动到函数的末尾。

```
def postorder(tree):
    if tree != None:
        postorder(tree.getLeftChild())
        postorder(tree.getRightChild())
        print(tree.getRootVal())
```

*Listing 4*

我们已经看到了后序遍历的常见用法，即计算分析树。再次回到 Listing 1。 我们所做的是计算左子树，计算右子树，并通过对操作符的函数调用在根节点中组合它们。假设我们的二叉树只存储表达式树的数据，让我们重写计算函数，需要更仔细地对 Listing 4中的后序遍历代码进行建模（参见 Listing 5）。

```
def postordereval(tree):
    opers = {'+':operator.add, '-':operator.sub, '*':operator.mul, '/':operator.truedi
v}
    res1 = None
    res2 = None
    if tree:
        res1 = postordereval(tree.getLeftChild())
        res2 = postordereval(tree.getRightChild())
        if res1 and res2:
            return opers[tree.getRootVal()](res1,res2)
        else:
            return tree.getRootVal()
```

*Listing 5*

请注意，Listing 4中的形式与 Listing 5中的形式相同，只是不在函数的末尾打印值，而是返回它。 这允许我们保存从第 6 行和第 7 行的递归调用返回的值。然后，我们使用这些保存的值以及第 9 行的运算符一起计算结果。

在本节中我们最终将看到中序遍历。 在中序遍历中，我们访问左子树，其次是根，最后是右子树。 Listing 6 展示了我们的中序遍历的代码。 注意，在所有三个遍历函数中，我们只是改变 print 语句相对于两个递归函数调用的位置。

```
def inorder(tree):
  if tree != None:
      inorder(tree.getLeftChild())
      print(tree.getRootVal())
      inorder(tree.getRightChild())
```

*Listing 6*

如果我们执行一个简单的中序遍历分析树，我们得到没有任何括号的原始表达式。 让我们修改基本的 `inorder` 算法，以允许我们恢复表达式的完全括号版本。 我们将对基本模板进行的唯一修改如下：在递归调用左子树之前打印左括号，并在递归调用右子树后打印右括号。 修改后的代码如 Listing 7所示。

```
def printexp(tree):
  sVal = ""
  if tree:
      sVal = '(' + printexp(tree.getLeftChild())
      sVal = sVal + str(tree.getRootVal())
      sVal = sVal + printexp(tree.getRightChild())+')'
  return sVal
```

# 6.8.基于二叉堆的优先队列

在前面的部分中，你了解了称为队列的先进先出数据结构。队列的一个重要变种称为优先级队列。优先级队列的作用就像一个队列，你可以通过从前面删除一个项目来出队。然而，在优先级队列中，队列中的项的逻辑顺序由它们的优先级确定。最高优先级项在队列的前面，最低优先级的项在后面。因此，当你将项排入优先级队列时，新项可能会一直移动到前面。我们将在下一章中研究一些图算法看到优先级队列是有用的数据结构。

你可能想到了几种简单的方法使用排序函数和列表实现优先级队列。然而，插入列表是 O(n) 并且排序列表是 O(nlogn)。我们可以做得更好。实现优先级队列的经典方法是使用称为二叉堆的数据结构。二叉堆将允许我们在 O(logn) 中排队和取出队列。

二叉堆是很有趣的研究，因为堆看起来很像一棵树，但是当我们实现它时，我们只使用一个单一的列表作为内部表示。二叉堆有两个常见的变体：最小堆（其中最小的键总是在前面）和最大堆（其中最大的键值总是在前面）。在本节中，我们将实现最小堆。我们将最大堆实现作为练习。

# 6.9.二叉堆操作

我们的二叉堆实现的基本操作如下：

- BinaryHeap() 创建一个新的，空的二叉堆。
- insert(k) 向堆添加一个新项。
- findMin() 返回具有最小键值的项，并将项留在堆中。
- delMin() 返回具有最小键值的项，从堆中删除该项。
- 如果堆是空的，isEmpty() 返回 true，否则返回 false。
- size() 返回堆中的项数。
- buildHeap(list) 从键列表构建一个新的堆。

ActiveCode 1 展示了使用一些二叉堆方法。注意，无论我们向堆中添加项的顺序是什么，每次都删除最小的。我们现在将把注意力转向如何实现这个想法。

```
from pythonds.trees.binheap import BinHeap

bh = BinHeap()
bh.insert(5)
bh.insert(7)
bh.insert(3)
bh.insert(11)

print(bh.delMin())

print(bh.delMin())

print(bh.delMin())

print(bh.delMin())
```

# 6.10.二叉堆实现

## 6.10.1.结构属性

为了使我们的堆有效地工作，我们将利用二叉树的对数性质来表示我们的堆。 为了保证对数性能，我们必须保持树平衡。平衡二叉树在根的左和右子树中具有大致相同数量的节点。 在我们的堆实现中，我们通过创建一个 `完整二叉树` 来保持树平衡。 一个完整的二叉树是一个树，其中每个层都有其所有的节点，除了树的最底层，从左到右填充。 Figure 1 展示了完整二叉树的示例。

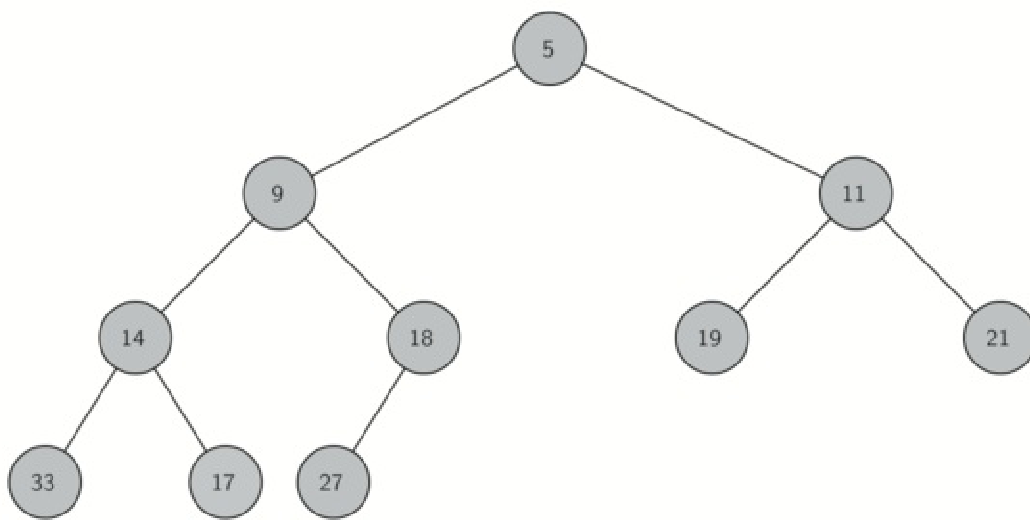



*Figure 1*

完整二叉树的另一个有趣的属性是，我们可以使用单个列表来表示它。 我们不需要使用节点和引用，甚至列表的列表。因为树是完整的，父节点的左子节点（在位置 p 处）是在列表中位置 2p 中找到的节点。 类似地，父节点的右子节点在列表中的位置 2p + 1。为了找到树中任意节点的父节点，我们可以简单地使用Python 的整数除法。 假定节点在列表中的位置 n，则父节点在位置 n/2。 Figure 2 展示了一个完整的二叉树，并给出了树的列表表示。 请注意父级和子级之间是 2p 和 2p+1 关系。 树的列表表示以及完整的结构属性允许我们仅使用几个简单的数学运算来高效地遍历一个完整的二叉树。 我们将看到，这也是我们的二叉堆的有效实现。

## 6.10.2.堆的排序属性

我们用于堆中存储项的方法依赖于维护堆的排序属性。 堆的排序属性如下：在堆中，对于具有父 p 的每个节点 x，p 中的键小于或等于 x 中的键。 Figure 2 展示了具有堆顺序属性的完整二叉树。

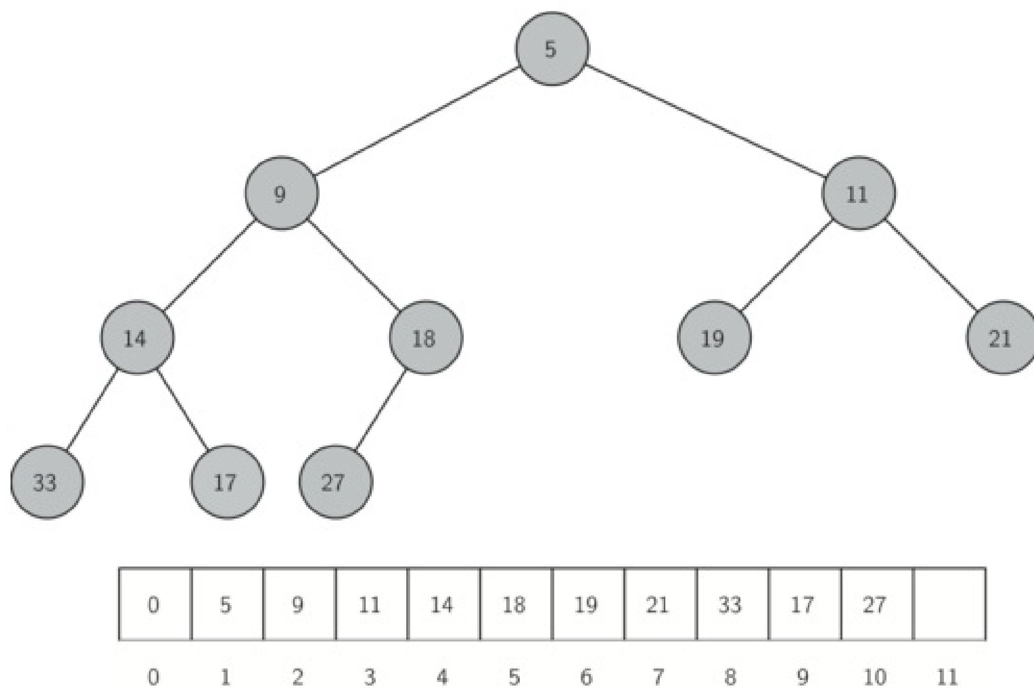



*Figure 2*

## 6.10.3.堆操作

我们将开始实现一个二叉堆的构造函数。由于整个二叉堆可以由单个列表表示，所以构造函数将初始化列表和一个 `currentSize` 属性来跟踪堆的当前大小。 Listing 1 展示了构造函数的 Python 代码。 你会注意到，一个空的二叉堆有一个单一的零作为 `heapList` 的第一个元素，这个零只是放那里，用于以后简单的整数除法。

```
class BinHeap:
    def __init__(self):
        self.heapList = [0]
        self.currentSize = 0
```

*Listing 1*

我们将实现的下一个方法是 `insert` 。 将项添加到列表中最简单，最有效的方法是将项附加到列表的末尾。 它维护完整的树属性。但可能违反堆结构属性。可以编写一个方法，通过比较新添加的项与其父项，我们可以重新获得堆结构属性。 如果新添加的项小于其父项，则我们可以将项与其父项交换。 Figure 2 展示了将新添加的项替换到其在树中的适当位置所需的操作。

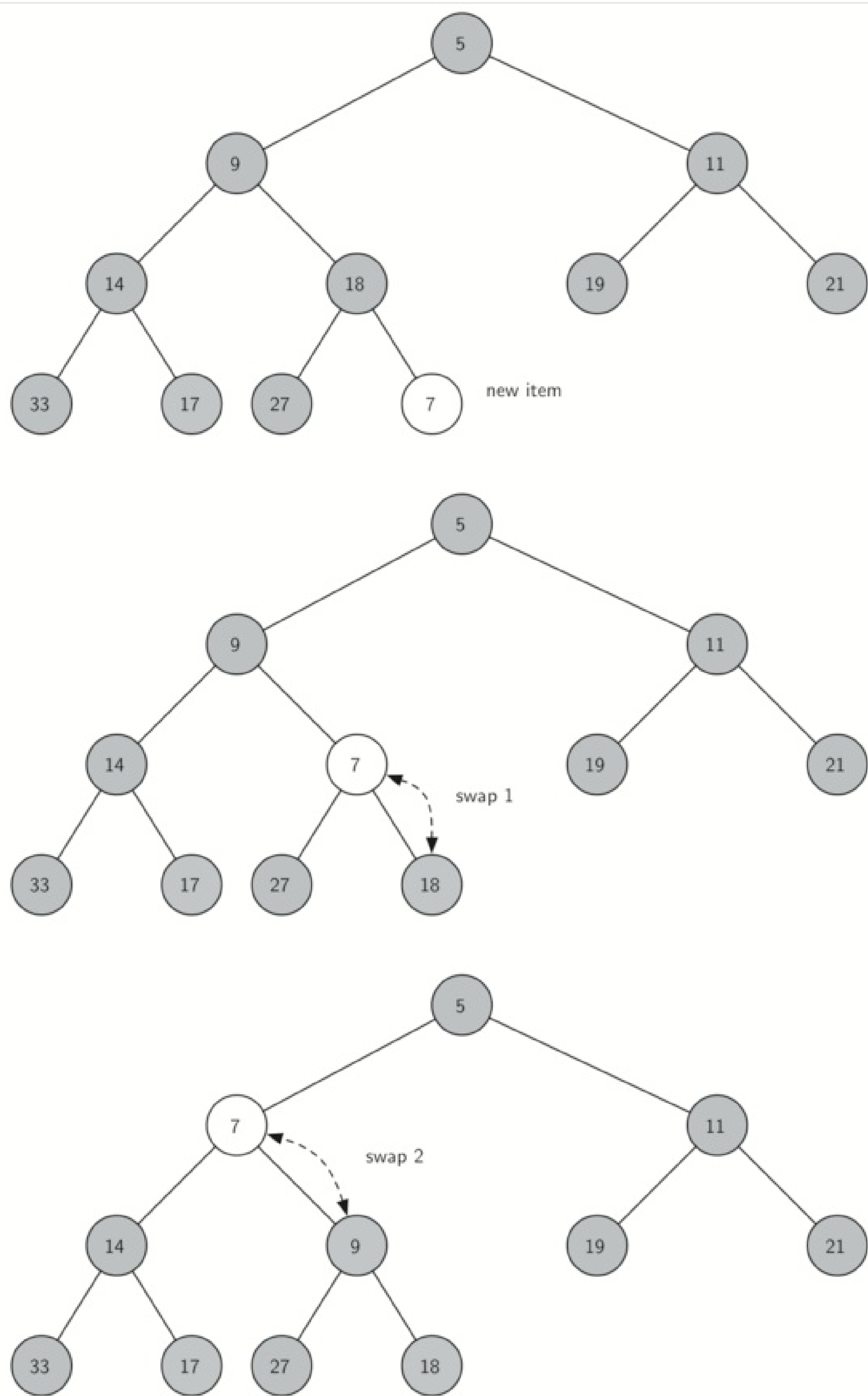



*Figure 2*

注意，当我们完成一个项时，我们需要恢复新添加的项和父项之间的堆属性。 我们还需保留任何兄弟节点的堆属性。当然，如果新添加的项非常小，我们可能仍需要将其交换另一上层。事实上，我们可能需要交换到树的顶部。 Listing 2 展示了 `percUp` 方法，它在树中向上

遍历一个新项，因为它需要去维护堆属性。 注意，我们可以通过使用简单的整数除法来计算任意节点的父节点。 当前节点的父节点可以通过将当前节点的索引除以 2 来计算。

我们现在可以编写 `insert` 方法了（见 Listing 3）。 插入方法中的大部分工作都是由 `percUp` 完成的。 一旦一个新项被追加到树上，`percUp` 接管并正确定位新项。

```
def percUp(self,i):
    while i // 2 > 0:
      if self.heapList[i] < self.heapList[i // 2]:
         tmp = self.heapList[i // 2]
         self.heapList[i // 2] = self.heapList[i]
         self.heapList[i] = tmp
      i = i // 2
```

*Listing 2*

```
def insert(self,k):
    self.heapList.append(k)
    self.currentSize = self.currentSize + 1
    self.percUp(self.currentSize)
```

*Listing 3*

使用正确定义的 `insert` 方法，我们现在可以看 `delMin` 方法。 因为堆属性要求树的根是树中的最小项，所以找到最小项很容易。`delMin` 的难点在根被删除后恢复堆结构和堆顺序属性。 我们可以分两步恢复我们的堆。首先，我们将通过获取列表中的最后一个项并将其移动到根位置来恢复根项，保持我们的堆结构属性。 但是，我们可能已经破坏了我们的二叉堆的

堆顺序属性。 第二，我们通过将新的根节点沿着树向下推到其正确位置来恢复堆顺序属性。 Figure 3展示了将新的根节点移动到堆中的正确位置所需的交换序列。 ee

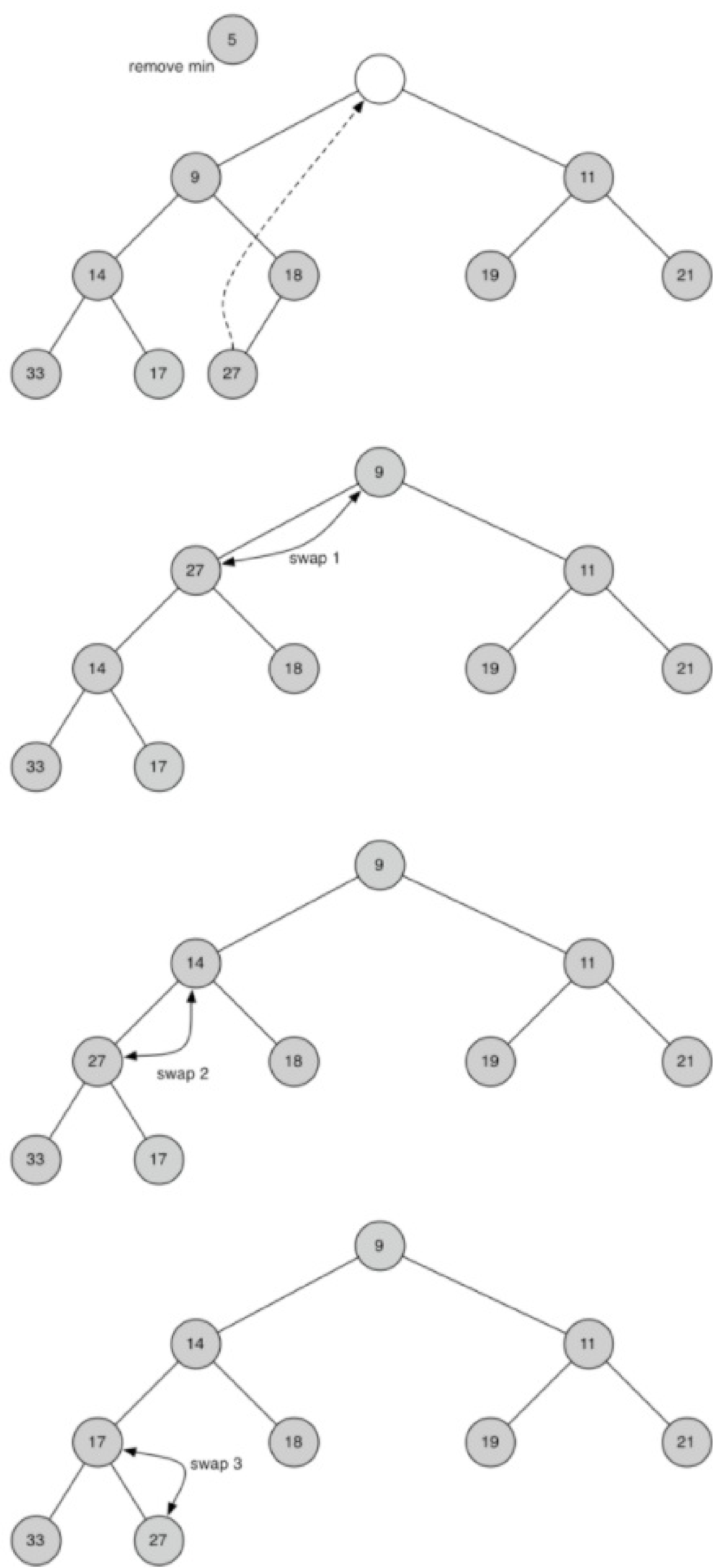

5
remove min
9
11
14
18
19
21
33
17
27
9
27
swap 1
11
14
18
19
21
33
17
9
14
11
27
swap 2
18
19
21
33
17
9
14
11
17
swap 3
18
19
21
33
27

*Figure 3*

为了维护堆顺序属性，我们所需要做的是将根节点和最小的子节点交换。在初始交换之后，我们可以将节点和其子节点重复交换，直到节点被交换到正确的位置，使它小于两个子节点。树交换节点的代码可以在 Listing 4中的 `percDown` 和 `minChild` 方法中找到。

```
def percDown(self,i):
    while (i * 2) <= self.currentSize:
        mc = self.minChild(i)
        if self.heapList[i] > self.heapList[mc]:
            tmp = self.heapList[i]
            self.heapList[i] = self.heapList[mc]
            self.heapList[mc] = tmp
        i = mc

def minChild(self,i):
    if i * 2 + 1 > self.currentSize:
        return i * 2
    else:
        if self.heapList[i*2] < self.heapList[i*2+1]:
            return i * 2
        else:
            return i * 2 + 1
```

*Listing 4*

`delmin` 操作的代码在 Listing 5 中。注意，有难度的工作由辅助函数处理，在这种情况下是 `percDown` 。

```
def delMin(self):
    retval = self.heapList[1]
    self.heapList[1] = self.heapList[self.currentSize]
    self.currentSize = self.currentSize - 1
    self.heapList.pop()
    self.percDown(1)
    return retval
```

*Listing 5*

为了完成我们对二叉堆的讨论，我们将看从一个列表构建整个堆的方法。你可能想到的第一种方法如下所示。给定一个列表，通过一次插入一个键轻松地构建一个堆。由于你从一个项的列表开始，该列表是有序的，可以使用二分查找找到正确的位置，以大约 O(log^n ) 操作的成本插入下一个键。 但是，请记住，在列表中间插入项可能需要 O(n) 操作来移动列表的其余部分，为新项腾出空间。 因此，要在堆中插入 n 个键，将需要总共 O(nlogn) 操作。 然而，如果我们从整个列表开始，那么我们可以在 O(n) 操作中构建整个堆。Listing 6 展示了构建整个堆的代码。

```
def buildHeap(self,alist):
    i = len(alist) // 2
    self.currentSize = len(alist)
    self.heapList = [0] + alist[:]
    while (i > 0):
        self.percDown(i)
        i = i - 1
```

*Listing 6*

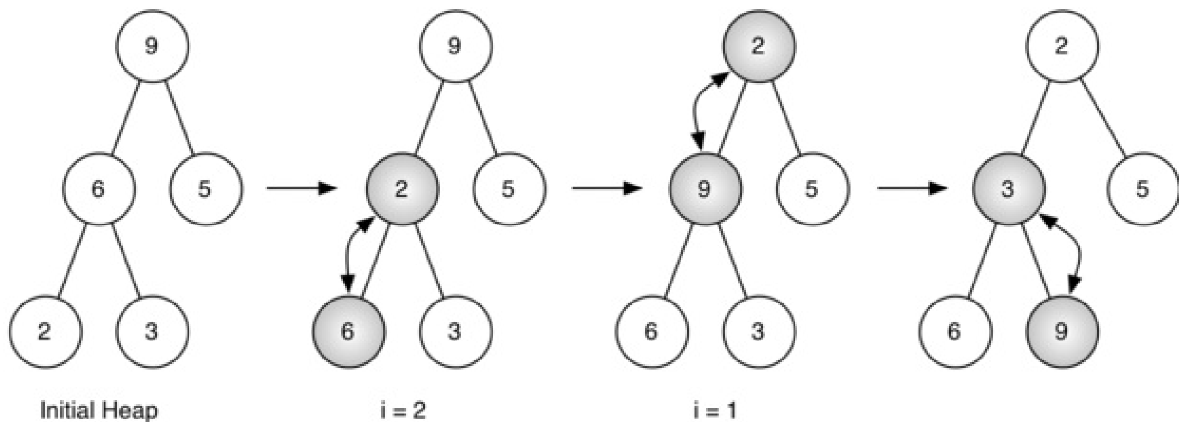



*Figure 4*

Figure 4 展示了 `buildHeap` 方法在 `[9,6,5,2,3]` 的初始树中的节点移动到其正确位置时所做的交换。虽然我们从树的中间开始，并以我们的方式回到根节点，`percDown` 方法确保最大的子节点总是沿着树向下移动。因为堆是一个完整的二叉树，超过中途点的任何节点都将是树叶，因此没有子节点。注意，当 `i = 1` 时，我们从树的根节点向下交换，因此可能需要多次交换。正如你在 Figure 4 最右边的两个树中可以看到的，首先 9 从根位置移出，但是 9 在树中向下移动一级之后，`percDown` 检查下一组子树，以确保它被推到下一层。在这种情况下，它与 3 进行第二次交换。现在 9 已经移动到树的最低层，不能进行进一步交换。将 Figure 4 所示的这一系列交换的列表与树进行比较是有用的。

```
i = 2  [0, 9, 5, 6, 2, 3]
i = 1  [0, 9, 2, 6, 5, 3]
i = 0  [0, 2, 3, 6, 5, 9]
```

完整二叉堆代码实现见 activecode 1

```
class BinHeap:
    def __init__(self):
        self.heapList = [0]
        self.currentSize = 0
```

```
    def percUp(self,i):
        while i // 2 > 0:
          if self.heapList[i] < self.heapList[i // 2]:
             tmp = self.heapList[i // 2]
             self.heapList[i // 2] = self.heapList[i]
             self.heapList[i] = tmp
          i = i // 2

    def insert(self,k):
      self.heapList.append(k)
      self.currentSize = self.currentSize + 1
      self.percUp(self.currentSize)

    def percDown(self,i):
      while (i * 2) <= self.currentSize:
          mc = self.minChild(i)
          if self.heapList[i] > self.heapList[mc]:
              tmp = self.heapList[i]
              self.heapList[i] = self.heapList[mc]
              self.heapList[mc] = tmp
          i = mc

    def minChild(self,i):
      if i * 2 + 1 > self.currentSize:
          return i * 2
      else:
          if self.heapList[i*2] < self.heapList[i*2+1]:
              return i * 2
          else:
              return i * 2 + 1

    def delMin(self):
      retval = self.heapList[1]
      self.heapList[1] = self.heapList[self.currentSize]
      self.currentSize = self.currentSize - 1
      self.heapList.pop()
      self.percDown(1)
      return retval

    def buildHeap(self,alist):
      i = len(alist) // 2
      self.currentSize = len(alist)
      self.heapList = [0] + alist[:]
      while (i > 0):
          self.percDown(i)
          i = i - 1

bh = BinHeap()
bh.buildHeap([9,5,6,2,3])

print(bh.delMin())
print(bh.delMin())
```

```
print(bh.delMin())
print(bh.delMin())
print(bh.delMin())
```

*ActiveCode 1*

我们可以在 O(n) 中构建堆的断言可能看起来有点神秘，证明超出了本书的范围。 然而，理解的关键是记住 log^n 因子是从树的高度派生的。 对于 `buildHeap` 中的大部分工作，树比 log^n 短。

基于可以从 O(n) 时间构建堆的事实，你可以使用堆对列表在 O(nlogn) 时间内排序，作为本章结尾的练习。

# 6.11.二叉查找树

我们已经看到了两种不同的方法来获取集合中的键值对。回想一下，这些集合实现了 `map` 抽象数据类型。我们讨论的 `map` ADT 的两个实现是在列表和哈希表上的二分搜索。在本节中，我们将研究二叉查找树作为从键映射到值的另一种方法。 在这种情况下，我们对树中项的确切位置不感兴趣，但我们有兴趣使用二叉树结构来提供高效的搜索。

# 6.12.查找树操作

在我们看实现之前，先来看看 map ADT 提供的接口。你会注意到，这个接口与Python 字典非常相似。

- Map() 创建一个新的空 `map` 。
- put(key，val) 向 `map` 中添加一个新的键值对。如果键已经在 `map` 中，那么用新值替换旧值。
- get(key) 给定一个键，返回存储在 `map` 中的值，否则为 None。
- del 使用 del map[key] 形式的语句从 `map` 中删除键值对。
- len() 返回存储在映射中的键值对的数量。
- in 返回 True 如果给定的键在 `map` 中。

# 6.13.查找树实现

二叉搜索树依赖于在左子树中找到的键小于父节点的属性，并且在右子树中找到的键大于父代。 我们将这个称为 bst属性。 当我们如上所述实现 Map 接口时，bst 属性将指导我们的实现。 Figure 1说明了二叉搜索树的此属性，展示了没有任何关联值的键。请注意，该属性适用于每个父级和子级。 左子树中的所有键小于根中的键。 右子树中的所有键都大于根。

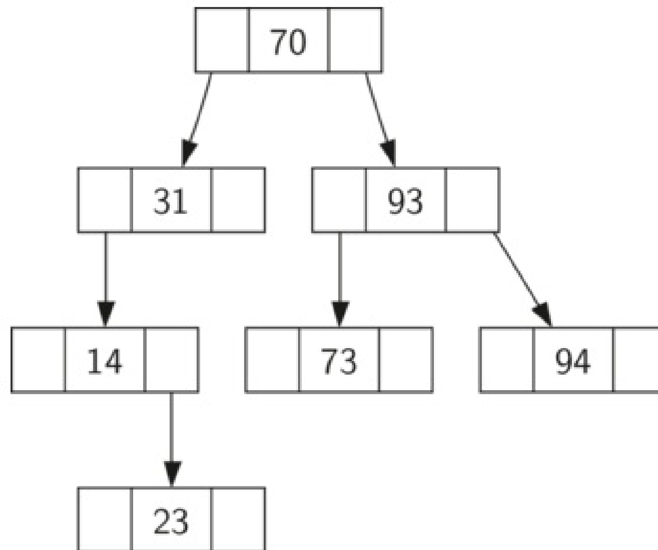



*Figure1*

现在你知道什么是二叉搜索树，我们将看看如何构造二叉搜索树。Figure 1中的搜索树表示在按照所示的顺序插入以下键之后存在的节点： `70,31,93,94,14,23,73` 。因为 70 是插入树中的第一个键，它是根。接下来，31 小于 70，所以它成为 70 的左孩子。接下来，93 大于 70，所以它成为 70 的右孩子。现在我们有两层的树填充，所以下一个键 94 ，因为 94 大于70 和 93，它成为 93 的右孩子。类似地，14 小于 70 和 31，所以它变成 31 的左孩子。23 也小于 31，所以它必须在左子树 31 中。但是，它大于14，所以它成为 14 的右孩子。

为了实现二叉搜索树，我们将使用类似于我们用于实现链表的节点和引用方法，以及表达式树。但是，因为我们必须能够创建和使用一个空的二叉搜索树，我们的实现将使用两个类。第一个类称为 `BinarySearchTree` ，第二个类称为 `TreeNode` 。 `BinarySearchTree` 类具有对作为二叉搜索树的根的 `TreeNode` 的引用。在大多数情况下，外部类中定义的外部方法只是检查树是否为空。如果树中有节点，请求只是传递到 `BinarySearchTree` 类中定义的私有方法，该方法以 root 作为参数。在树是空的或者我们想要删除树根的键的情况下，我们必须采取特殊的行动。 `BinarySearchTree` 类构造函数的代码以及一些其他杂项函数如Listing 1所示。

```
class BinarySearchTree:

    def __init__(self):
        self.root = None
        self.size = 0

    def length(self):
        return self.size

    def __len__(self):
        return self.size

    def __iter__(self):
        return self.root.__iter__()
```

*Listing 1*

TreeNode 类提供了许多辅助函数，使得在 `BinarySearchTree` 类方法中完成的工作更容易。`TreeNode` 的构造函数以及这些辅助函数如 Listing 2所示。你可以在列表中看到许多辅助函数根据自己的位置将节点分类为子节点（左或右）和节点具有的子节点类型。 `TreeNode` 类还将显式地跟踪父节点作为每个节点的属性。当我们讨论 del 操作符的实现时，你会看到为什么这很重要。

Listing 2 中 `TreeNode` 另一个有趣的方面是我们使用 Python 的可选参数。可选参数使我们能够在几种不同的情况下轻松创建 `TreeNode` 。有时我们想要构造一个已经同时具有父和子的新 `TreeNode` 。对于现有的父和子，我们可以传递父和子作为参数。在其他时候，我们将使用键值对创建一个`TreeNode` ，我们不会为父或子传递任何参数。在这种情况下，将使用可选参数的默认值。

```
class TreeNode:
   def __init__(self,key,val,left=None,right=None,
                                       parent=None):
        self.key = key
        self.payload = val
        self.leftChild = left
        self.rightChild = right
        self.parent = parent

    def hasLeftChild(self):
        return self.leftChild

    def hasRightChild(self):
        return self.rightChild

    def isLeftChild(self):
        return self.parent and self.parent.leftChild == self

    def isRightChild(self):
        return self.parent and self.parent.rightChild == self

    def isRoot(self):
        return not self.parent

    def isLeaf(self):
        return not (self.rightChild or self.leftChild)

    def hasAnyChildren(self):
        return self.rightChild or self.leftChild

    def hasBothChildren(self):
        return self.rightChild and self.leftChild

    def replaceNodeData(self,key,value,lc,rc):
        self.key = key
        self.payload = value
        self.leftChild = lc
        self.rightChild = rc
        if self.hasLeftChild():
            self.leftChild.parent = self
        if self.hasRightChild():
            self.rightChild.parent = self
```

*Listing 2*

现在我们有了 `BinarySearchTree` shell 和 `TreeNode` ，现在是时候编写 `put` 方法，这将允许我们构建二叉搜索树。 `put` 方法是 `BinarySearchTree` 类的一个方法。此方法将检查树是否已具有根。如果没有根，那么 `put` 将创建一个新的 `TreeNode` 并将其做为树的根。如果根节点已经就位，则 put 调用私有递归辅助函数 `_put` 根据以下算法搜索树：

- 从树的根开始，搜索二叉树，将新键与当前节点中的键进行比较。如果新键小于当前节点，则搜索左子树。如果新键大于当前节点，则搜索右子树。
- 当没有左（或右）孩子要搜索时，我们在树中找到应该建立新节点的位置。
- 要向树中添加节点，请创建一个新的 `TreeNode` 对象，并将对象插入到上一步发现的节点。

Listing 3 展示了在树中插入一个新节点的 Python 代码。 `_put` 函数按照上述步骤递归编写。请注意，当一个新的子节点插入到树中时， `currentNode` 将作为父节点传递给新的树节点。

我们实现插入的一个重要问题是重复的键不能正确处理。当我们的树被实现时，重复键将在具有原始键的节点的右子树中创建具有相同键值的新节点。这样做的结果是，具有新键的节点将永远不会在搜索期间被找到。处理插入重复键的更好方法是将新键相关联的值替换旧值。我们将修复这个bug作为练习。

```
def put(self,key,val):
    if self.root:
        self._put(key,val,self.root)
    else:
        self.root = TreeNode(key,val)
    self.size = self.size + 1

def _put(self,key,val,currentNode):
    if key < currentNode.key:
        if currentNode.hasLeftChild():
               self._put(key,val,currentNode.leftChild)
        else:
               currentNode.leftChild = TreeNode(key,val,parent=currentNode)
    else:
        if currentNode.hasRightChild():
               self._put(key,val,currentNode.rightChild)
        else:
               currentNode.rightChild = TreeNode(key,val,parent=currentNode)
```

*Listing 3*

当 `put` 方法定义后，我们可以通过使用 `__setitem__` 方法调用（参见 Listing 4 ） `put` 方法来重载赋值的 `[]` 运算符。这使得我们可以编写像 `myZipTree['Plymouth'] = 55446` 这样的 Python 语句，就像 Python 字典一样。

```
def __setitem__(self,k,v):
    self.put(k,v)
```

*Listing 4*

Figure 2 展示了用于将新节点插入二叉搜索树的过程。 浅阴影的节点指示在插入过程期间访问的节点。

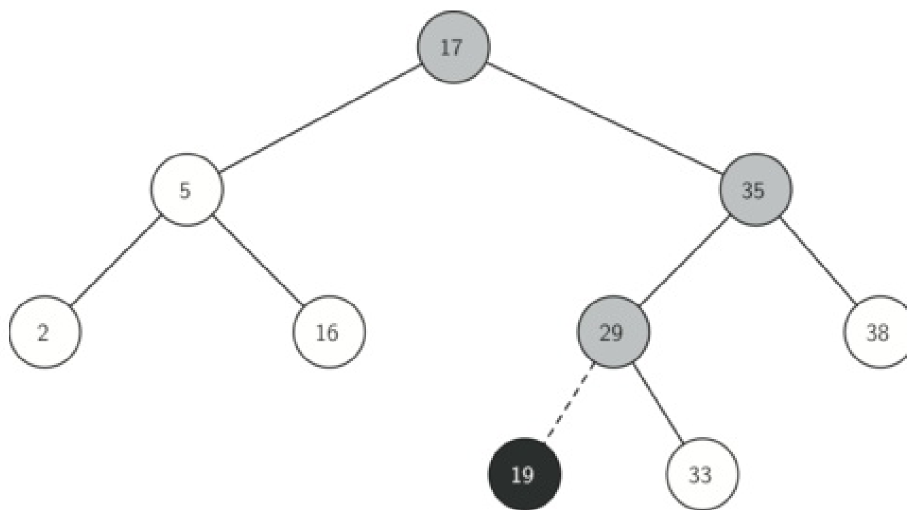



*Figure 2*

一旦树被构造，下一个任务是实现对给定键的值的检索。 `get` 方法比 `put` 方法更容易，因为它只是递归地搜索树，直到它到达不匹配的叶节点或找到匹配的键。当找到匹配的键时，返回存储在节点的有效载荷中的值。

Listing 5 展示了 `get` ， `_get` 和 `__getitem__` 的代码。 `_get` 方法中的搜索代码使用和 `_put` 相同的逻辑来选择左或右子节点。请注意， `_get` 方法返回一个 `TreeNode` ，这允许 `_get` 用作其他 `BinarySearchTree` 方法的一个灵活的辅助方法，可能需要利用除了有效载荷之外的 `TreeNode` 的其他数据。

通过实现 `__getitem__` 方法，我们可以编写一个类似于访问字典的 Python 语句，而实际上我们使用的是二叉搜索树，例如 `z = myZipTree ['Fargo']` 。正如你所看到的，所有的 `__getitem__` 方法都是调用 `get` 。

```
def get(self,key):
    if self.root:
        res = self._get(key,self.root)
        if res:
               return res.payload
        else:
               return None
    else:
        return None

def _get(self,key,currentNode):
    if not currentNode:
        return None
    elif currentNode.key == key:
        return currentNode
    elif key < currentNode.key:
        return self._get(key,currentNode.leftChild)
    else:
        return self._get(key,currentNode.rightChild)

def __getitem__(self,key):
    return self.get(key)
```

*Listing 5*

使用 `get` ，我们可以通过为 `BinarySearchTree` 写一个 `__contains__` 方法来实现 in 操作。 `__contains__` 方法将简单地调用 `get` 并在 `get` 返回值时返回 True，如果返回 None 则返回 False。 `__contains__` 的代码如Listing 6所示。

```
def __contains__(self,key):
    if self._get(key,self.root):
        return True
    else:
        return False
```

*Listing 6*

回想一下， `__contains__` 重载了 `in` 操作符，允许我们写出如下语句：

```
if 'Northfield' in myZipTree:
    print("oom ya ya")
```

最后，我们将注意力转向二叉搜索树中最具挑战性的方法，删除一个键（参见 Listing 7）。第一个任务是通过搜索树来找到要删除的节点。如果树具有多个节点，我们使用 `_get` 方法搜索以找到需要删除的 `TreeNode` 。如果树只有一个节点，这意味着我们删除树的根，但是

我们仍然必须检查以确保根的键匹配要删除的键。 在任一情况下，如果未找到键， `del` 操作符将引发错误。

```
def delete(self,key):
   if self.size > 1:
      nodeToRemove = self._get(key,self.root)
      if nodeToRemove:
          self.remove(nodeToRemove)
          self.size = self.size-1
      else:
          raise KeyError('Error, key not in tree')
   elif self.size == 1 and self.root.key == key:
      self.root = None
      self.size = self.size - 1
   else:
      raise KeyError('Error, key not in tree')

def __delitem__(self,key):
    self.delete(key)
```

*Listing 7*

一旦我们找到了我们要删除的键的节点，我们必须考虑三种情况：

1. 要删除的节点没有子节点（参见Figure 3）。
2. 要删除的节点只有一个子节点（见Figure 4）。
3. 要删除的节点有两个子节点（见Figure 5）。

第一种情况很简单（见 Listing 8）。 如果当前节点没有子节点，我们需要做的是删除节点并删除对父节点中该节点的引用。 此处的代码如下所示。

```
if currentNode.isLeaf():
    if currentNode == currentNode.parent.leftChild:
        currentNode.parent.leftChild = None
    else:
        currentNode.parent.rightChild = None
```

*Listing 8*

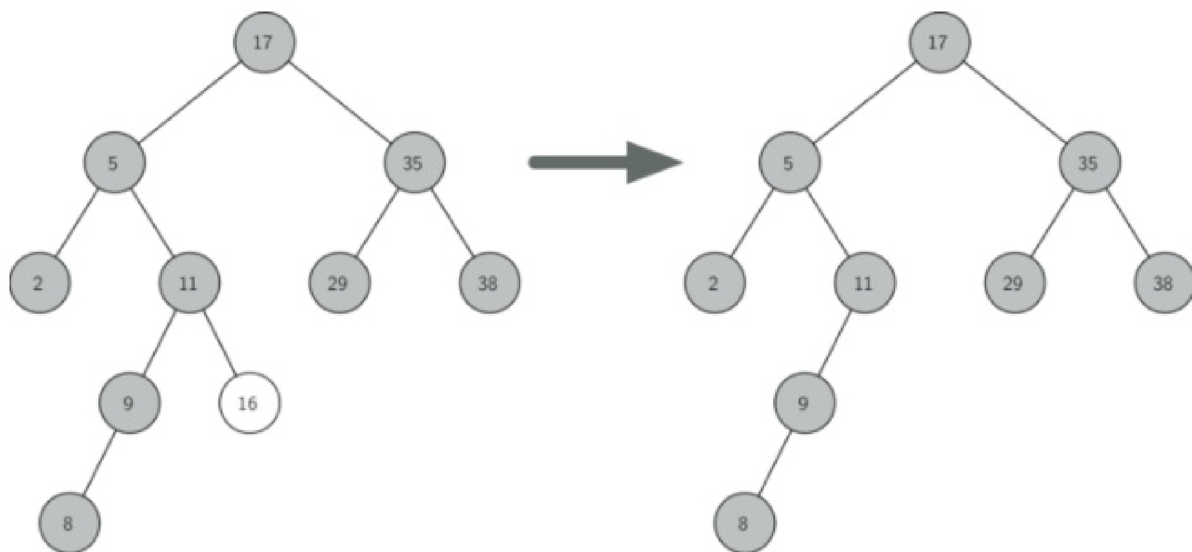



*Figure 3*

第二种情况只是稍微复杂一点（见 Listing 9）。如果一个节点只有一个孩子，那么我们可以简单地促进孩子取代其父。此案例的代码展示在下一个列表中。当你看这个代码，你会看到有六种情况要考虑。由于这些情况相对于左孩子或右孩子对称，我们将仅讨论当前节点具有左孩子的情况。决策如下：

1. 如果当前节点是左子节点，则我们只需要更新左子节点的父引用以指向当前节点的父节点，然后更新父节点的左子节点引用以指向当前节点的左子节点。
2. 如果当前节点是右子节点，则我们只需要更新左子节点的父引用以指向当前节点的父节点，然后更新父节点的右子节点引用以指向当前节点的左子节点。
3. 如果当前节点没有父级，则它是根。在这种情况下，我们将通过在根上调用 `replaceNodeData` 方法来替换 `key`，`payload`，`leftChild` 和 `rightChild` 数据。

```
else: # this node has one child
   if currentNode.hasLeftChild():
      if currentNode.isLeftChild():
          currentNode.leftChild.parent = currentNode.parent
          currentNode.parent.leftChild = currentNode.leftChild
      elif currentNode.isRightChild():
          currentNode.leftChild.parent = currentNode.parent
          currentNode.parent.rightChild = currentNode.leftChild
      else:
          currentNode.replaceNodeData(currentNode.leftChild.key,
                             currentNode.leftChild.payload,
                             currentNode.leftChild.leftChild,
                             currentNode.leftChild.rightChild)
   else:
      if currentNode.isLeftChild():
          currentNode.rightChild.parent = currentNode.parent
          currentNode.parent.leftChild = currentNode.rightChild
      elif currentNode.isRightChild():
          currentNode.rightChild.parent = currentNode.parent
          currentNode.parent.rightChild = currentNode.rightChild
      else:
          currentNode.replaceNodeData(currentNode.rightChild.key,
                             currentNode.rightChild.payload,
                             currentNode.rightChild.leftChild,
                             currentNode.rightChild.rightChild)
```

*Listing 9*

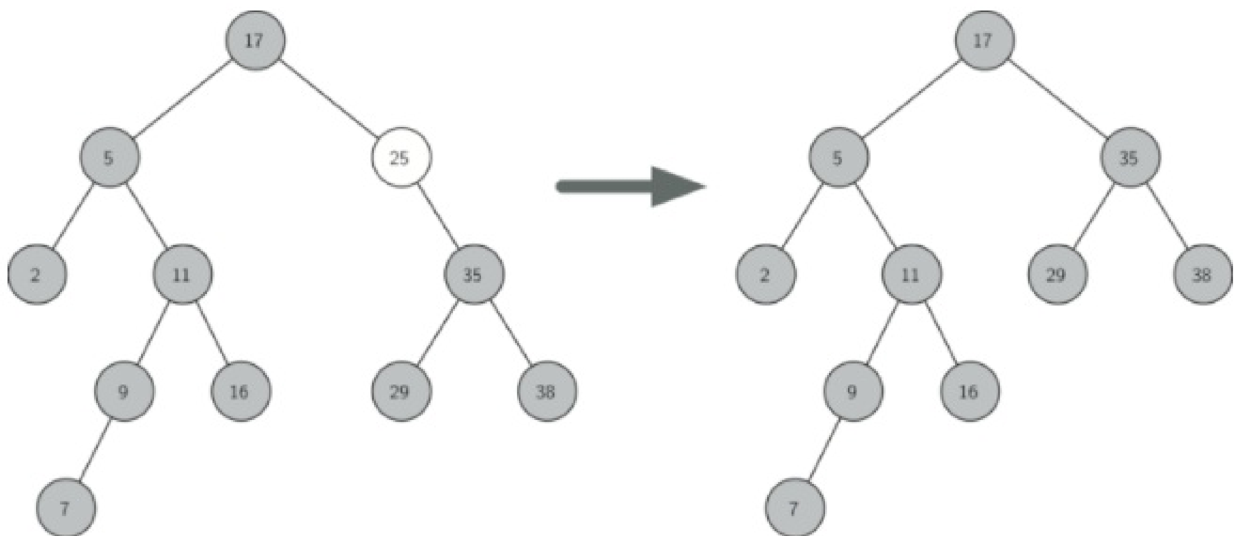



*Figure 4*

第三种情况是最难处理的情况（见Listing 10）。如果一个节点有两个孩子，那么我们不太可能简单地提升其中一个节点来占据节点的位置。然而，我们可以在树中搜索可用于替换被调度删除的节点的节点。我们需要的是一个节点，它将保留现有的左和右子树的二叉搜索树关

系。 执行此操作的节点是树中具有次最大键的节点。 我们将这个节点称为后继节点，我们将看一种方法来很快找到后继节点。 继承节点保证没有多于一个孩子，所以我们知道使用已经实现的两种情况删除它。 一旦删除了后继，我们只需将它放在树中，代替要删除的节点。

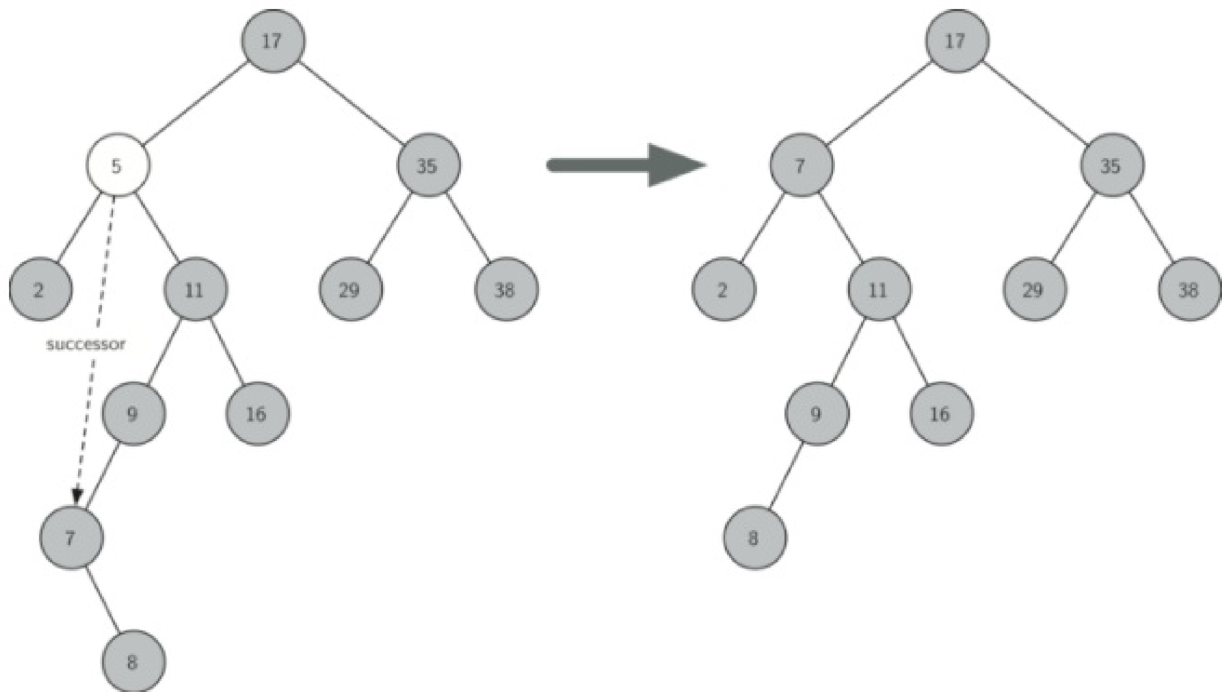



*Figure 5*

处理第三种情况的代码展示在下一个列表中。 注意，我们使用辅助方法 `findSuccessor` 和 `findMin` 来找到后继。 要删除后继，我们使用 `spliceOut` 方法。 我们使用 `spliceOut` 的原因是它直接找到我们想要拼接的节点，并做出正确的更改。 我们可以递归调用删除，但是我们将浪费时间重新搜索关键节点。

```
elif currentNode.hasBothChildren(): #interior
        succ = currentNode.findSuccessor()
        succ.spliceOut()
        currentNode.key = succ.key
        currentNode.payload = succ.payload
```

*Listing 10*

找到后继的代码如下所示（见 Listing 11），是 `TreeNode` 类的一个方法。此代码利用二叉搜索树的相同属性，采用中序遍历从最小到最大打印树中的节点。在寻找接班人时，有三种情况需要考虑：

1. 如果节点有右子节点，则后继节点是右子树中的最小的键。
2. 如果节点没有右子节点并且是父节点的左子节点，则父节点是后继节点。
3. 如果节点是其父节点的右子节点，并且它本身没有右子节点，则此节点的后继节点是其父节点的后继节点，不包括此节点。

第一个条件是对于我们从二叉搜索树中删除节点时唯一重要的条件。但是，`findSuccessor` 方法具有其他用法，我们将在本章结尾的练习中介绍。

调用 `findMin` 方法来查找子树中的最小键。你应该说服自己，任何二叉搜索树中的最小值键是树的最左子节点。因此，`findMin` 方法简单地循环子树的每个节点中的 `leftChild` 引用，直到它到达没有左子节点的节点。

```
def findSuccessor(self):
    succ = None
    if self.hasRightChild():
        succ = self.rightChild.findMin()
    else:
        if self.parent:
               if self.isLeftChild():
                   succ = self.parent
               else:
                   self.parent.rightChild = None
                   succ = self.parent.findSuccessor()
                   self.parent.rightChild = self
    return succ

def findMin(self):
    current = self
    while current.hasLeftChild():
        current = current.leftChild
    return current

def spliceOut(self):
    if self.isLeaf():
        if self.isLeftChild():
               self.parent.leftChild = None
        else:
               self.parent.rightChild = None
    elif self.hasAnyChildren():
        if self.hasLeftChild():
               if self.isLeftChild():
                  self.parent.leftChild = self.leftChild
               else:
                  self.parent.rightChild = self.leftChild
               self.leftChild.parent = self.parent
        else:
               if self.isLeftChild():
                  self.parent.leftChild = self.rightChild
               else:
                  self.parent.rightChild = self.rightChild
               self.rightChild.parent = self.parent
```

*Listing 11*

我们需要查看二叉搜索树的最后一个接口方法。假设我们想要按中序遍历树中的所有键。我们肯定用字典做，为什么不是树？你已经知道如何使用中序遍历算法按顺序遍历二叉树。然而，编写迭代器需要更多的工作，因为迭代器在每次调用迭代器时只返回一个节点。

Python 为我们提供了一个非常强大的函数，在创建迭代器时使用。该函数称为 `yield` 。 `yield` 类似于 `return` ，因为它向调用者返回一个值。然而，yield 采取冻结函数状态的附加步骤，使得下一次调用函数时，它从其早先停止的确切点继续执行。创建可以迭代的对象的函数称为生成函数。

二叉树的 `inorder` 迭代器的代码展示在下一个列表中。仔细看看这段代码;乍一看，你可能认为代码不是递归的。但是，请记住， `__iter__` 覆盖 `for x in` 操作，因此它是递归的！因为它是在 `TreeNode` 实例上递归的，所以 `__iter__` 方法在 `TreeNode` 类中定义。

```
def __iter__(self):
   if self:
      if self.hasLeftChild():
             for elem in self.leftChild:
                yield elem
      yield self.key
      if self.hasRightChild():
             for elem in self.rightChild:
                yield elem
```

此时，你可能需要下载包含完整版本的 `BinarySearchTree` 和 `TreeNode` 类的整个文件。

```
class TreeNode:
    def __init__(self,key,val,left=None,right=None,parent=None):
        self.key = key
        self.payload = val
        self.leftChild = left
        self.rightChild = right
        self.parent = parent

    def hasLeftChild(self):
        return self.leftChild

    def hasRightChild(self):
        return self.rightChild

    def isLeftChild(self):
        return self.parent and self.parent.leftChild == self

    def isRightChild(self):
        return self.parent and self.parent.rightChild == self

    def isRoot(self):
        return not self.parent

    def isLeaf(self):
```

```
        return not (self.rightChild or self.leftChild)

    def hasAnyChildren(self):
        return self.rightChild or self.leftChild

    def hasBothChildren(self):
        return self.rightChild and self.leftChild

    def replaceNodeData(self,key,value,lc,rc):
        self.key = key
        self.payload = value
        self.leftChild = lc
        self.rightChild = rc
        if self.hasLeftChild():
            self.leftChild.parent = self
        if self.hasRightChild():
            self.rightChild.parent = self


class BinarySearchTree:

    def __init__(self):
        self.root = None
        self.size = 0

    def length(self):
        return self.size

    def __len__(self):
        return self.size

    def put(self,key,val):
        if self.root:
            self._put(key,val,self.root)
        else:
            self.root = TreeNode(key,val)
        self.size = self.size + 1

    def _put(self,key,val,currentNode):
        if key < currentNode.key:
            if currentNode.hasLeftChild():
                   self._put(key,val,currentNode.leftChild)
            else:
                   currentNode.leftChild = TreeNode(key,val,parent=currentNode)
        else:
            if currentNode.hasRightChild():
                   self._put(key,val,currentNode.rightChild)
            else:
                   currentNode.rightChild = TreeNode(key,val,parent=currentNode)

    def __setitem__(self,k,v):
       self.put(k,v)
```

```
    def get(self,key):
       if self.root:
           res = self._get(key,self.root)
           if res:
                  return res.payload
           else:
                  return None
       else:
           return None

    def _get(self,key,currentNode):
       if not currentNode:
           return None
       elif currentNode.key == key:
           return currentNode
       elif key < currentNode.key:
           return self._get(key,currentNode.leftChild)
       else:
           return self._get(key,currentNode.rightChild)

    def __getitem__(self,key):
       return self.get(key)

    def __contains__(self,key):
       if self._get(key,self.root):
           return True
       else:
           return False

    def delete(self,key):
      if self.size > 1:
         nodeToRemove = self._get(key,self.root)
         if nodeToRemove:
             self.remove(nodeToRemove)
             self.size = self.size-1
         else:
             raise KeyError('Error, key not in tree')
      elif self.size == 1 and self.root.key == key:
         self.root = None
         self.size = self.size - 1
      else:
         raise KeyError('Error, key not in tree')

    def __delitem__(self,key):
       self.delete(key)

    def spliceOut(self):
       if self.isLeaf():
           if self.isLeftChild():
                  self.parent.leftChild = None
           else:
                  self.parent.rightChild = None
       elif self.hasAnyChildren():
```

```
        if self.hasLeftChild():
               if self.isLeftChild():
                  self.parent.leftChild = self.leftChild
               else:
                  self.parent.rightChild = self.leftChild
               self.leftChild.parent = self.parent
        else:
               if self.isLeftChild():
                  self.parent.leftChild = self.rightChild
               else:
                  self.parent.rightChild = self.rightChild
               self.rightChild.parent = self.parent

def findSuccessor(self):
  succ = None
  if self.hasRightChild():
      succ = self.rightChild.findMin()
  else:
      if self.parent:
             if self.isLeftChild():
                 succ = self.parent
             else:
                 self.parent.rightChild = None
                 succ = self.parent.findSuccessor()
                 self.parent.rightChild = self
  return succ

def findMin(self):
  current = self
  while current.hasLeftChild():
      current = current.leftChild
  return current

def remove(self,currentNode):
     if currentNode.isLeaf(): #leaf
       if currentNode == currentNode.parent.leftChild:
           currentNode.parent.leftChild = None
       else:
           currentNode.parent.rightChild = None
     elif currentNode.hasBothChildren(): #interior
       succ = currentNode.findSuccessor()
       succ.spliceOut()
       currentNode.key = succ.key
       currentNode.payload = succ.payload

     else: # this node has one child
       if currentNode.hasLeftChild():
         if currentNode.isLeftChild():
             currentNode.leftChild.parent = currentNode.parent
             currentNode.parent.leftChild = currentNode.leftChild
         elif currentNode.isRightChild():
             currentNode.leftChild.parent = currentNode.parent
             currentNode.parent.rightChild = currentNode.leftChild
```

```
            else:
                currentNode.replaceNodeData(currentNode.leftChild.key,
                                   currentNode.leftChild.payload,
                                   currentNode.leftChild.leftChild,
                                   currentNode.leftChild.rightChild)
          else:
            if currentNode.isLeftChild():
                currentNode.rightChild.parent = currentNode.parent
                currentNode.parent.leftChild = currentNode.rightChild
            elif currentNode.isRightChild():
                currentNode.rightChild.parent = currentNode.parent
                currentNode.parent.rightChild = currentNode.rightChild
            else:
                currentNode.replaceNodeData(currentNode.rightChild.key,
                                   currentNode.rightChild.payload,
                                   currentNode.rightChild.leftChild,
                                   currentNode.rightChild.rightChild)




mytree = BinarySearchTree()
mytree[3]="red"
mytree[4]="blue"
mytree[6]="yellow"
mytree[2]="at"

print(mytree[6])
print(mytree[2])
```

# 6.14.查找树分析

随着二叉搜索树的实现完成，我们将对已经实现的方法进行快速分析。让我们先来看看 `put` 方法。其性能的限制因素是二叉树的高度。从词汇部分回忆一下树的高度是根和最深叶节点之间的边的数量。高度是限制因素，因为当我们寻找合适的位置将一个节点插入到树中时，我们需要在树的每个级别最多进行一次比较。

二叉树的高度可能是多少？这个问题的答案取决于如何将键添加到树。如果按照随机顺序添加键，树的高度将在 log2^n 附近，其中 n 是树中的节点数。这是因为如果键是随机分布的，其中大约一半将小于根，一半大于根。请记住，在二叉树中，根节点有一个节点，下一级节点有两个节点，下一个节点有四个节点。任何特定级别的节点数为 2^d ，其中 d 是级别的深度。完全平衡的二叉树中的节点总数为 2^h+1 - 1，其中 h 表示树的高度。

完全平衡的树在左子树中具有与右子树相同数量的节点。在平衡二叉树中，`put` 的最坏情况性能是 O(log2^n ），其中 n 是树中的节点数。注意，这是与前一段中的计算的反比关系。所以 log2^n 给出了树的高度，并且表示了在适当的位置插入新节点时，需要做的最大比较次数。

不幸的是，可以通过以排序顺序插入键来构造具有高度 n 的搜索树！这样的树的示例见 Figure 6。在这种情况下，put方法的性能是 O(n)。

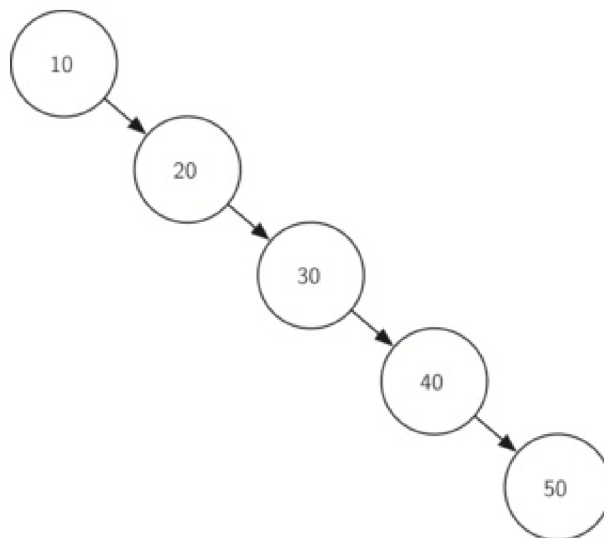



*Figure 6*

现在你明白了 `put` 方法的性能受到树的高度的限制，你可能猜测其他方法 `get` ，`in` 和 `del` 也是有限制的。 由于 `get` 搜索树以找到键，在最坏的情况下，树被一直搜索到底部，并且没有找到键。 乍一看，`del` 似乎更复杂，因为它可能需要在删除操作完成之前搜索后继。 但请记住，找到后继者的最坏情况也只是树的高度，这意味着你只需要加倍工作。 因为加倍是一个常数因子，它不会改变最坏的情况

# 6.15.平衡二叉搜索树

在上一节中，我们考虑构建一个二叉搜索树。正如我们所学到的，二叉搜索树的性能可以降级到 O(n) 的操作，如 `get` 和 `put` ，如果树变得不平衡。在本节中，我们将讨论一种特殊类型的二叉搜索树，它自动确保树始终保持平衡。这棵树被称为 AVL树，以其发明人命名：G.M. Adelson-Velskii 和E.M.Landis。

AVL树实现 Map 抽象数据类型就像一个常规的二叉搜索树，唯一的区别是树的执行方式。为了实现我们的 AVL树，我们需要跟踪树中每个节点的平衡因子。我们通过查看每个节点的左右子树的高度来做到这一点。更正式地，我们将节点的平衡因子定义为左子树的高度和右子树的高度之间的差。

> balanceFactor = height(leftSubTree) - height(rightSubTree)

使用上面给出的平衡因子的定义，我们说如果平衡因子大于零，则子树是左重的。如果平衡因子小于零，则子树是右重的。如果平衡因子是零，那么树是完美的平衡。为了实现AVL树，并且获得具有平衡树的好处，如果平衡因子是 -1,0 或 1，我们将定义树平衡。一旦树中的节点的平衡因子是在这个范围之外，我们将需要一个程序来使树恢复平衡。Figure 1展示了不平衡，右重树和每个节点的平衡因子的示例。

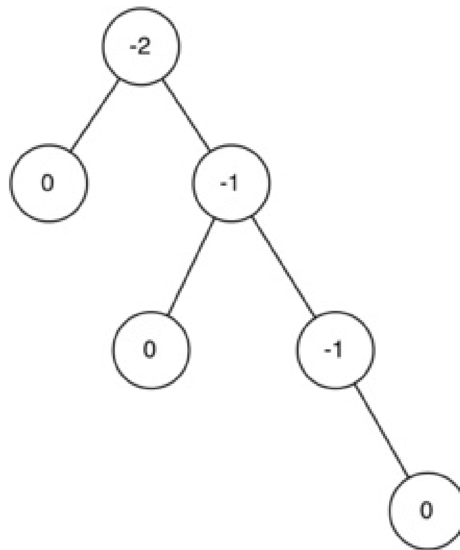



*Figure 1*

# 6.16.AVL平衡二叉搜索树

在我们继续之前，我们来看看执行这个新的平衡因子要求的结果。我们的主张是，通过确保树总是具有 -1,0或1 的平衡因子，我们可以获得更好的操作性能的关键操作。 让我们开始思考这种平衡条件如何改变最坏情况的树。有两种可能性，一个左重树和一个右重树。 如果我们考虑高度0,1,2和3的树，Figure 2 展示了在新规则下可能的最不平衡的左重树。

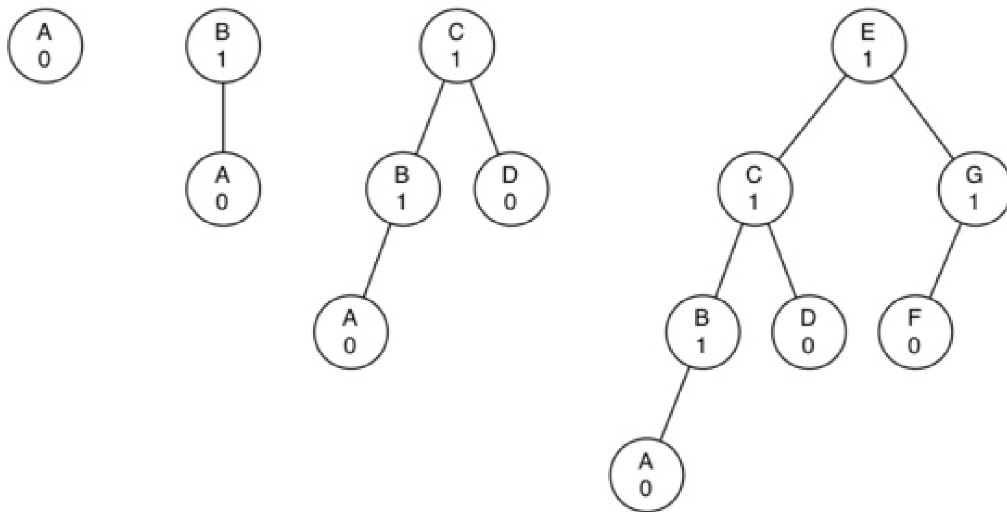



*Figure 2*

看树中节点的总数，我们看到对于高度为0的树，有1个节点，对于高度为1的树，有1 + 1 = 2个节点，对于高度为2的树 是1 + 1 + 2 = 4，对于高度为3的树，有1 + 2 + 4 = 7。 更一般地，我们看到的高度h(Nh) 的树中的节点数量的模式是：

$$N_h = 1 + N_{h-1} + N_{h-2}$$

这种可能看起来很熟悉，因为它非常类似于斐波纳契序列。 给定树中节点的数量，我们可以使用这个事实来导出AVL树的高度的公式。 回想一下，对于斐波纳契数列，第i个斐波纳契数字由下式给出：

$$F_0 = 0$$
$$F_1 = 1$$
$$F_i = F_{i-1} + F_{i-2} \text{ for all } i \geq 2$$

一个重要的数学结果是，随着斐波纳契数列越来越大，$F_i/F_{i-1}$ 的比率越来越接近黄金比率 `Φ= (1 +√5)/2` 。 如果要查看上一个方程的导数，可以查阅数学文本。 我们将简单地使用该方程来近似 Fi，如 Fi =Φ^i / 5。 如果我们利用这个近似，我们可以重写 Nh 的方程为：

$$N_h = F_{h+2} - 1, h \geq 1$$

通过用其黄金比例近似替换斐波那契参考，我们得到：

$$N_h = \frac{\Phi^{h+2}}{\sqrt{5}} - 1$$

如果我们重新排列这些项，并取两边的底数为2的对数，然后求解 h，我们得到以下推导：

$$\log N_h + 1 = (H + 2)\log \Phi - \frac{1}{2}\log 5$$
$$h = \frac{\log N_h + 1 - 2\log \Phi + \frac{1}{2}\log 5}{\log \Phi}$$
$$h = 1.44 \log N_h$$

这个推导告诉我们，在任何时候，我们的AVL树的高度等于树中节点数目的对数的常数（1.44）倍。 这是搜索我们的AVL树的好消息，因为它将搜索限制为O(logN）。

# 6.17.AVL平衡二叉搜索树实现

现在我们已经证明保持 AVL树的平衡将是一个很大的性能改进，让我们看看如何增加过程来插入一个新的键到树。由于所有新的键作为叶节点插入到树中，并且我们知道新叶的平衡因子为零，所以刚刚插入的节点没有新的要求。但一旦添加新叶，我们必须更新其父的平衡因子。这个新叶如何影响父的平衡因子取决于叶节点是左孩子还是右孩子。如果新节点是右子节点，则父节点的平衡因子将减少1。如果新节点是左子节点，则父节点的平衡因子将增加1。这个关系可以递归地应用到新节点的祖父节点，并且应用到每个祖先一直到树的根。由于这是一个递归过程，我们来看一下用于更新平衡因子的两种基本情况：

- 递归调用已到达树的根。
- 父节点的平衡因子已调整为零。你应该说服自己，一旦一个子树的平衡因子为零，那么它的祖先节点的平衡不会改变。

我们将实现 AVL 树作为 `BinarySearchTree` 的子类。首先，我们将覆盖 `_put` 方法并编写一个新的 `updateBalance` 辅助方法。这些方法如Listing 1所示。你将注意到，`_put` 的定义与简单二叉搜索树中的完全相同，除了第 7 行和第 13 行上对 `updateBalance` 的调用的添加。

```
def _put(self,key,val,currentNode):
    if key < currentNode.key:
        if currentNode.hasLeftChild():
                self._put(key,val,currentNode.leftChild)
        else:
                currentNode.leftChild = TreeNode(key,val,parent=currentNode)
                self.updateBalance(currentNode.leftChild)
    else:
        if currentNode.hasRightChild():
                self._put(key,val,currentNode.rightChild)
        else:
                currentNode.rightChild = TreeNode(key,val,parent=currentNode)
                self.updateBalance(currentNode.rightChild)

def updateBalance(self,node):
    if node.balanceFactor > 1 or node.balanceFactor < -1:
        self.rebalance(node)
        return
    if node.parent != None:
        if node.isLeftChild():
                node.parent.balanceFactor += 1
        elif node.isRightChild():
                node.parent.balanceFactor -= 1

        if node.parent.balanceFactor != 0:
                self.updateBalance(node.parent)
```

*Listing 1*

新的 `updateBalance` 方法完成了大多数工作。这实现了我们刚才描述的递归过程。`updateBalance` 方法首先检查当前节点是否不够平衡，需要重新平衡（第16行）。如果平衡，则重新平衡完成，并且不需要对父节点进行进一步更新。如果当前节点不需要重新平衡，则调整父节点的平衡因子。如果父的平衡因子不为零，那么算法通过递归调用父对象上的 `updateBalance`，继续沿树向根向上运行。

当需要树重新平衡时，我们如何做呢？有效的重新平衡是使AVL树在不牺牲性能的情况下正常工作的关键。为了使AVL树恢复平衡，我们将在树上执行一个或多个旋转。

要理解旋转是什么让我们看一个非常简单的例子。考虑 Figure 3左半部分的树。这棵树平衡因子为 -2，不平衡。为了使这棵树平衡，我们将使用以节点 A 为根的子树的左旋转。

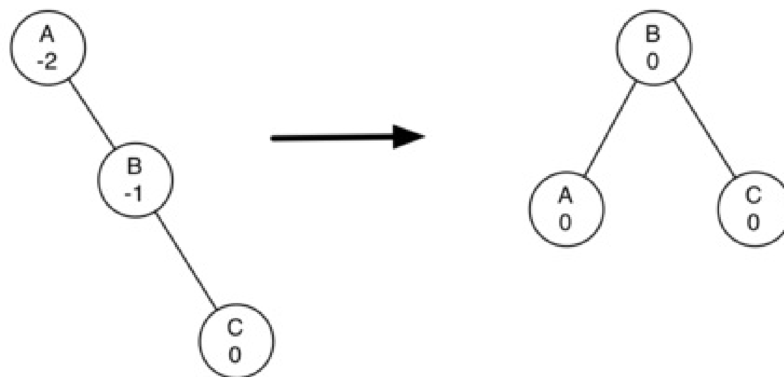



*Figure 3*

要执行左旋转，我们基本上执行以下操作：

- 提升右孩子（B）成为子树的根。
- 将旧根（A）移动为新根的左子节点。
- 如果新根（B）已经有一个左孩子，那么使它成为新左孩子（A）的右孩子。注意：由于新根（B）是A的右孩子，A 的右孩子在这一点上保证为空。这允许我们添加一个新的节点作为右孩子，不需进一步的考虑。

虽然这个过程在概念上相当容易，但是代码的细节有点棘手，因为我们需要按照正确的顺序移动事物，以便保留二叉搜索树的所有属性。此外，我们需要确保适当地更新所有的父指针。

让我们看一个稍微更复杂的树来说明右旋转。Figure 4的左侧显示了树的左重，在根处的平衡因子为 2。要执行右旋转，我们基本上执行以下操作：

- 提升左子节点（C）为子树的根。
- 将旧根（E）移动为新根的右子树。
- 如果新根（C）已经有一个正确的孩子（D），那么使它成为新的右孩子（E）的左孩

子。注意：由于新根（C）是 E 的左子节点，因此 E 的左子节点在此时保证为空。这允许我们添加一个新节点作为左孩子，不需进一步的考虑。

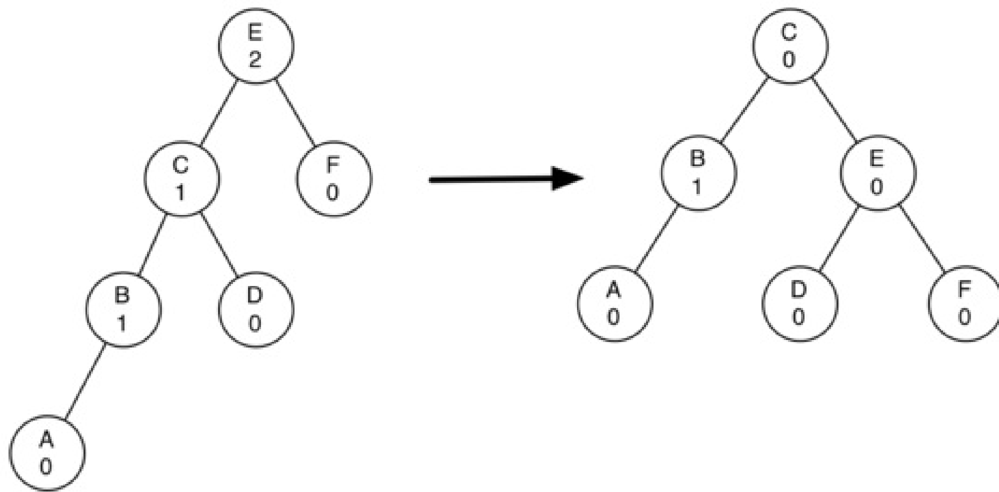



*Figure 4*

现在你已经看到了旋转，并且有旋转的工作原理的基本概念，让我们看看代码。Listing 2展示了右旋转和左旋转的代码。在第2行中，我们创建一个临时变量来跟踪子树的新根。正如我们之前所说的，新的根是上一个根的右孩子。现在对这个临时变量存储了一个对右孩子的引用，我们用新的左孩子替换旧根的右孩子。

下一步是调整两个节点的父指针。如果 newRoot 有一个左子节点，那么左子节点的新父节点变成旧的根节点。新根的父节点设置为旧根的父节点。如果旧根是整个树的根，那么我们必须设置树的根以指向这个新根。否则，如果旧根是左孩子，则我们将左孩子的父节点更改为指向新根;否则我们改变右孩子的父亲指向新的根。（行10-13）。最后，我们将旧根的父节点设置为新根。这是一个很复杂的过程，所以我们鼓励你跟踪这个功能，同时看下 Figure 3。`rotateRight` 方法是对称的 `rotateLeft` ，所以我们将留给你来研究 `rotateRight` 的代码。

```
def rotateLeft(self,rotRoot):
    newRoot = rotRoot.rightChild
    rotRoot.rightChild = newRoot.leftChild
    if newRoot.leftChild != None:
        newRoot.leftChild.parent = rotRoot
    newRoot.parent = rotRoot.parent
    if rotRoot.isRoot():
        self.root = newRoot
    else:
        if rotRoot.isLeftChild():
                rotRoot.parent.leftChild = newRoot
        else:
            rotRoot.parent.rightChild = newRoot
    newRoot.leftChild = rotRoot
    rotRoot.parent = newRoot
    rotRoot.balanceFactor = rotRoot.balanceFactor + 1 - min(newRoot.balanceFactor, 0)
    newRoot.balanceFactor = newRoot.balanceFactor + 1 + max(rotRoot.balanceFactor, 0)
```

*Listing 2*

最后，第16-17行需要一些解释。 在这两行中，我们更新旧根和新根的平衡因子。 由于所有其他移动都是移动整个子树，所以所有其他节点的平衡因子不受旋转的影响。 但是我们如何在不完全重新计算新子树的高度的情况下更新平衡因子呢？ 以下推导应该能说服你这些行是正确的。

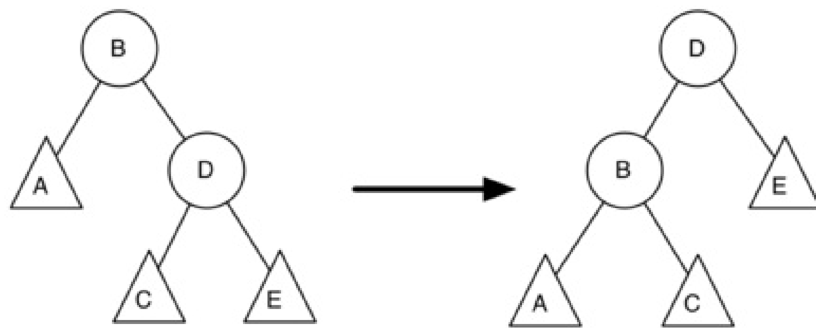



*Figure 5*

Figure 5 展示了左旋转。 B 和 D 是关键节点，A，C，E 是它们的子树。 设hx 表示以节点 x 为根的特定子树的高度。 根据定义，我们知道以下：

> newBal(B)=hA−hC oldBal(B)=hA−hD

但我们知道，D 的旧高度也可以由 `1 + max(hC，hE)` 给出，也就是说，D 的高度比其两个孩子的最大高度大 1。 记住， `hC` 和 `hE` 没有改变。 所以，让我们用第二个方程来代替它

> oldBal(B)=hA−(1+max(hC,hE))

然后减去这两个方程。 以下步骤进行减法并使用一些代数来简化 `newBal(B)` 的等式。

> $newBal(B)-oldBal(B)=hA-hC-(hA-(1+max(hC,hE)))$ $newBal(B)-oldBal(B)=hA-hC-hA+(1+max(hC,hE))$ $newBal(B)-oldBal(B)=hA-hA+1+max(hC,hE)-hC$ $newBal(B)-oldBal(B)=1+max(hC,hE)-hC$

接下来我们将 oldBal(B) 移动到方程的右边，并利用 `max(a，b) -c = max(a-c，b-c)` 。

> $newBal(B)=oldBal(B)+1+max(hC-hC,hE-hC)$

但是，hE-hC 与 -oldBal(D) 相同。因此，我们可以使用另一个表示 max(-a，-b) = -min(a，b) 的标识。 因此，我们可以完成我们的 newBal(B) 的推导，具有以下步骤：

> $newBal(B)=oldBal(B)+1+max(0,-oldBal(D))$ $newBal(B)=oldBal(B)+1-min(0,oldBal(D))$

现在我们有所有的部分，我们很容易知道。 我们记住 B 是 rotRoot 和 D 是newRoot 然后我们可以看到这正好对应第16行的语句，或者：

```
rotRoot.balanceFactor = rotRoot.balanceFactor + 1 - min(0,newRoot.balanceFactor)
```

类似的推导给出了更新的节点 D 的方程，以及右旋转后的平衡因子。 我们把这些作为你的练习。

现在你可能认为我们已经完成了。 我们知道如何做左右旋转，我们知道什么时候应该做左旋或右旋，但是看看 Figure 6。由于节点 A 的平衡因子为-2，我们应该做左旋转。 但是，当我们围绕A做左旋转时会发生什么？

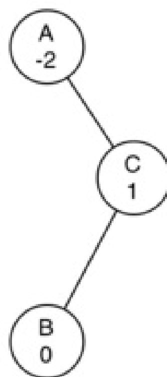



*Figure 6*

Figure 7 展示了我们在左旋后，我们现在已经在另一方面失去平衡。 如果我们做右旋以纠正这种情况，我们就回到我们开始的地方。

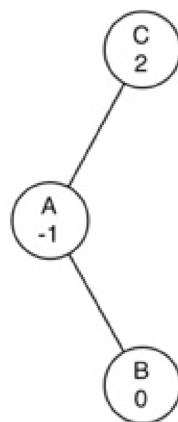



*Figure 7*

要纠正这个问题，我们必须使用以下规则集：

- 如果子树需要左旋转使其平衡，首先检查右子节点的平衡因子。 如果右孩子是重的，那么对右孩子做右旋转，然后是原来的左旋转。
- 如果子树需要右旋转使其平衡，首先检查左子节点的平衡因子。 如果左孩子是重的，那么对左孩子做左旋转，然后是原来的右旋转。

Figure 8展示了这些规则如何解决我们在Figure 6和 Figure 7中遇到的困境。从围绕节点 C的右旋转开始，将树放置在 A 的左旋转使整个子树恢复平衡的位置。

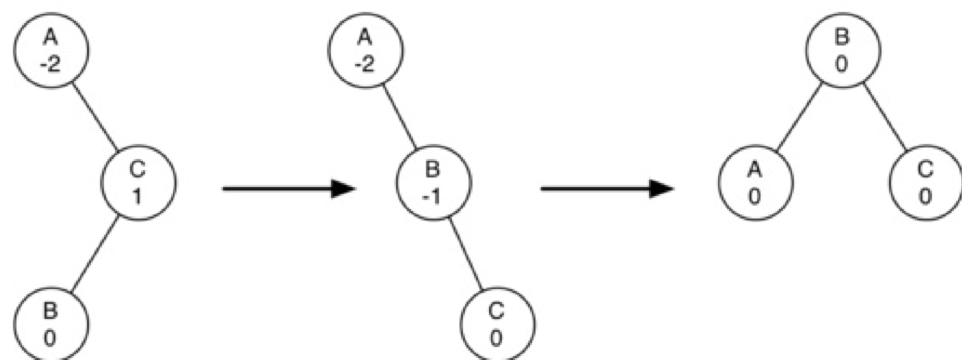



*Figure 8*

实现这些规则的代码可以在我们的重新平衡方法中找到，如 Listing 3所示。上面的规则编号 1 是从第2行开始的if语句实现的。规则编号2是由第8行开始的elif语句实现的 。

```
def rebalance(self,node):
  if node.balanceFactor < 0:
         if node.rightChild.balanceFactor > 0:
            self.rotateRight(node.rightChild)
            self.rotateLeft(node)
         else:
            self.rotateLeft(node)
  elif node.balanceFactor > 0:
         if node.leftChild.balanceFactor < 0:
            self.rotateLeft(node.leftChild)
            self.rotateRight(node)
         else:
            self.rotateRight(node)
```

*Listing 3*

通过保持树在所有时间的平衡，我们可以确保 get 方法将按 O(log2(n)) 时间运行。但问题是我们的 put 方法有什么成本？让我们将它分解为 put 执行的操作。由于将新节点作为叶子插入，更新所有父节点的平衡因子将需要最多log2^n 运算，树的每层一个运算。如果发现子树不平衡，则需要最多两次旋转才能使树重新平衡。但是，每个旋转在 O(1)时间中工作，因此我们的put操作仍然是O（log2^n )。

在这一点上，我们已经实现了一个功能AVL树，除非你需要删除一个节点的能力。我们保留删除节点和随后的更新和重新平衡作为一个练习。

# 6.18.Map抽象数据结构总结

在前面两章中，我们已经研究了可以用于实现 Map 抽象数据类型的几个数据结构。二叉搜索表，散列表，二叉搜索树和平衡二叉搜索树。 总结这一节，让我们总结 Map ADT 定义的关键操作的每个数据结构的性能（见Table 1）。

| operation | Sorted List | Hash Table | Binary Search Tree | AVL Tree |
|---|---|---|---|---|
| put | $O(n)$ | $O(1)$ | $O(n)$ | $O(\log_2 n)$ |
| get | $O(\log_2 n)$ | $O(1)$ | $O(n)$ | $O(\log_2 n)$ |
| in | $O(\log_2 n)$ | $O(1)$ | $O(n)$ | $O(\log_2 n)$ |
| del | $O(n)$ | $O(1)$ | $O(n)$ | $O(\log_2 n)$ |

# 6.19.总结

在这一章中，我们看了树的数据结构。 树数据结构使我们能够编写许多有趣的算法。 在本章中，我们研究了使用树来执行以下操作的算法：

- 用于解析和计算表达式的二叉树。
- 用于实现 Map ADT的二叉树。
- 用于实现 Map ADT的平衡二叉树（AVL树）。
- 一个二叉树实现一个最小堆。
- 用于实现优先级队列的最小堆。

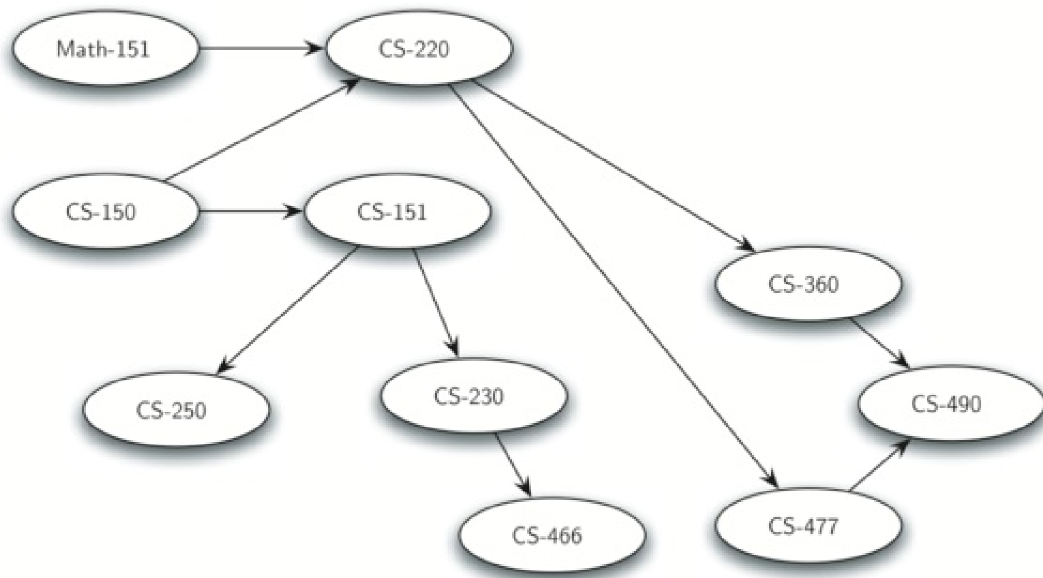
# 7.1.目标

- 了解图是什么，以及如何使用它。
- 使用多个内部表示来实现图抽象数据类型。
- 看看如何使用图来解决各种各样的问题

在本章中，我们将研究图。图是比我们在上一章中研究的树更通用的结构;实际上你可以认为树是一种特殊的图。图可以用来表示我们世界上许多有趣的事情，包括道路系统，从城市到城市的航空公司航班，互联网如何连接，甚至是完成计算机科学专业必须完成的课程顺序。我们将在本章中看到，一旦我们有一个问题的好的表示，我们可以使用一些标准图算法来解决，否则可能是一个非常困难的问题。

虽然人们相对容易看路线图并且理解不同地点之间的关系，但计算机没有这样的知识。然而，我们也可以将路线图视为图。当我们这样做时，我们可以让我们的计算机为我们做有趣的事情。如果你曾经使用过一个互联网地图网站，你知道一台计算机可以找到从一个地方到另一个地方最短，最快或最简单的路径。

作为计算机科学的学生，你可能想知道你必须学习的课程，以获得一个学位。图是表示学该课程之前的先决条件和其他相互依存关系的好方法。Figure 1 展示了另一个图。这个代表了在路德学院完成计算机科学专业的课程和顺序。



# 7.2.词汇和定义

现在我们已经看了一些图的示例，我们将更正式地定义图及其组件。我们已经从对树的讨论中知道了一些术语。

顶点 顶点（也称为“节点”）是图的基本部分。它可以有一个名称，我们将称为“键”。一个顶点也可能有额外的信息。我们将这个附加信息称为“有效载荷”。

边 边（也称为“弧”）是图的另一个基本部分。边连接两个顶点，以表明它们之间存在关系。边可以是单向的或双向的。如果图中的边都是单向的，我们称该图是 `有向图` 。上面显示的课程先决条件显然是一个图，因为你必须在其他课程之前学习一些课程。

权重 边可以被加权以示出从一个顶点到另一个顶点的成本。例如，在将一个城市连接到另一个城市的道路的图表中，边上的权重可以表示两个城市之间的距离。

利用这些定义，我们可以正式定义图。图可以由 G 表示，其中 `G =(V，E)` 。对于图 G，V 是一组顶点，E 是一组边。每个边是一个元组 `(v，w)` ，其中 `w,v ∈ V` 。我们可以添加第三个组件到边元组来表示权重。子图 s 是边 e 和顶点 v 的集合，使得 `e⊂E` 和 `v⊂V` 。

Figure 2 展示了简单加权有向图的另一示例。正式地，我们可以将该图表示为六个顶点的集合：

**V={V0,V1,V2,V3,V4,V5}**

和 9 条边的集合

**E={(v0,v1,5),(v1,v2,4),(v2,v3,9),(v3,v4,7),(v4,v0,1),(v0,v5,2),(v5,v4,8),(v3,v5,3),(v5,v2,1)}**

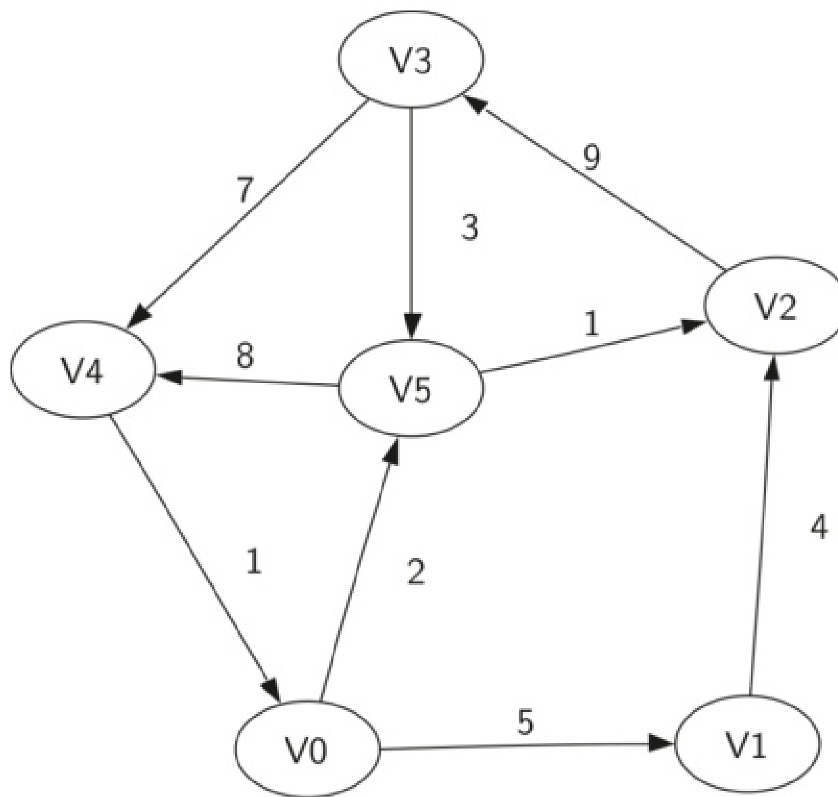



*Figure 2*

figure 2 中的示例图有助于说明两个其他关键图形术语：

路径 图中的路径是由边连接的顶点序列。形式上，我们将定义一个路径为 `w1，w2，...，wn` ，使得 `(wi，wi + 1) ∈ E` , 当 `1≤i≤ n-1` 。未加权路径长度是路径中的边的数目，具体是 `n-1` 。加权路径长度是路径中所有边的权重的总和。例如在 Figure 2中，从 V3 到 V1 的路径是顶点序列 `(V3，V4，V0，V1)` 。边是 `{（v3，v4,7），（v4，v0,1），（v0，v1,5）} }` 。

循环 有向图中的循环是在同一顶点开始和结束的路径。例如，在 Figure 2中，路径 `（V5，V2，V3，V5）` 是一个循环。没有循环的图形称为非循环图形。没有循环的有向图称为有向无环图或 `DAG` 。我们将看到，如果问题可以表示为 `DAG` ，我们可以解决几个重要的问题。

# 7.3.图抽象数据类型

图抽象数据类型（ADT）定义如下：

- Graph() 创建一个新的空图。
- addVertex(vert) 向图中添加一个顶点实例。
- addEdge(fromVert, toVert) 向连接两个顶点的图添加一个新的有向边。
- addEdge(fromVert, toVert, weight) 向连接两个顶点的图添加一个新的加权的有向边。
- getVertex(vertKey) 在图中找到名为 `vertKey` 的顶点。
- getVertices() 返回图中所有顶点的列表。
- in 返回 True 如果 `vertex in graph` 里给定的顶点在图中，否则返回False。

从图的正式定义开始，我们有几种方法可以在 Python 中实现图 ADT。 我们将看到在使用不同的表示来实现上述 ADT 时存在权衡。 有两个众所周知的图形、实现，`邻接矩阵` 和 `邻接表` 。 我们将解释这两个选项，然后实现一个作为 Python 类。

# 7.4.邻接矩阵

实现图的最简单的方法之一是使用二维矩阵。在该矩阵实现中，每个行和列表示图中的顶点。存储在行 v 和列 w 的交叉点处的单元中的值表示是否存在从顶点 v 到顶点 w 的边。 当两个顶点通过边连接时，我们说它们是相邻的。 Figure 3 展示了 Figure 2 中的图的邻接矩阵。单元格中的值表示从顶点 v 到顶点 w 的边的权重。

| | V0 | V1 | V2 | V3 | V4 | V5 |
|---|---|---|---|---|---|---|
| V0 | | 5 | | | | 2 |
| V1 | | | 4 | | | |
| V2 | | | | 9 | | |
| V3 | | | | | 7 | 3 |
| V4 | 1 | | | | | |
| V5 | | | 1 | | 8 | |

*Figure 3*

邻接矩阵的优点是简单，对于小图，很容易看到哪些节点连接到其他节点。 然而，注意矩阵中的大多数单元格是空的。 因为大多数单元格是空的，我们说这个矩阵是“稀疏的”。矩阵不是一种非常有效的方式来存储稀疏数据。 事实上，在Python中，你甚至要创建一个如 Figure 3所示的矩阵结构。

当边的数量大时，邻接矩阵是图的良好实现。但是什么是大？填充矩阵需要多少边？ 由于图中每个顶点有一行和一列，填充矩阵所需的边数为 |V|^2。 当每个顶点连接到每个其他顶点时，矩阵是满的。有几个真实的问题，接近这种连接。 我们在本章中讨论的问题都涉及稀疏连接的图。

# 7.5.邻接表

实现稀疏连接图的更空间高效的方法是使用邻接表。在邻接表实现中，我们保存Graph 对象中的所有顶点的主列表，然后图中的每个顶点对象维护连接到的其他顶点的列表。 在我们的顶点类的实现中，我们将使用字典而不是列表，其中字典键是顶点，值是权重。 Figure 4 展示了 Figure 2中的图的邻接列表表示。

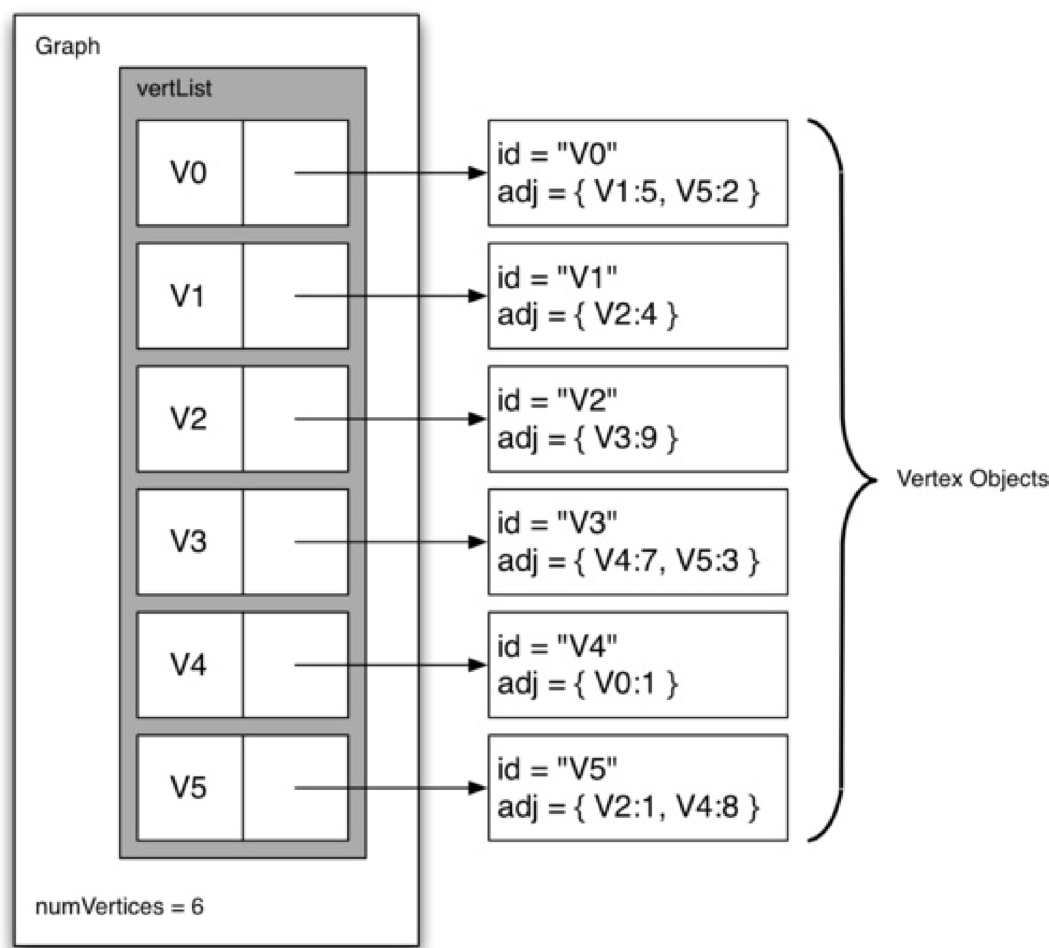



*Figure 4*

邻接表实现的优点是它允许我们紧凑地表示稀疏图。 邻接表还允许我们容易找到直接连接到特定顶点的所有链接。

# 7.6.实现

使用字典，很容易在 Python 中实现邻接表。在我们的 Graph 抽象数据类型的实现中，我们将创建两个类（见 Listing 1和 Listing 2），Graph（保存顶点的主列表）和 Vertex（将表示图中的每个顶点）。

每个顶点使用字典来跟踪它连接的顶点和每个边的权重。这个字典称为 `connectedTo` 。下面的列表展示了 `Vertex` 类的代码。构造函数只是初始化 `id` ，通常是一个字符串和 `connectedTo` 字典。 `addNeighbor` 方法用于从这个顶点添加一个连接到另一个。 `getConnections` 方法返回邻接表中的所有顶点，如 `connectedTo` 实例变量所示。 `getWeight` 方法返回从这个顶点到作为参数传递的顶点的边的权重。

```
class Vertex:
    def __init__(self,key):
        self.id = key
        self.connectedTo = {}

    def addNeighbor(self,nbr,weight=0):
        self.connectedTo[nbr] = weight

    def __str__(self):
        return str(self.id) + ' connectedTo: ' + str([x.id for x in self.connectedTo])

    def getConnections(self):
        return self.connectedTo.keys()

    def getId(self):
        return self.id

    def getWeight(self,nbr):
        return self.connectedTo[nbr]
```

*Listing 1*

下一个列表中显示的 Graph 类包含将顶点名称映射到顶点对象的字典。在 Figure 4中，该字典对象由阴影灰色框表示。 `Graph` 还提供了将顶点添加到图并将一个顶点连接到另一个顶点的方法。 `getVertices` 方法返回图中所有顶点的名称。此外，我们实现了 `__iter__` 方法，以便轻松地遍历特定图中的所有顶点对象。 这两种方法允许通过名称或对象本身在图形中的顶点上进行迭代。

```
class Graph:
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0

    def addVertex(self,key):
        self.numVertices = self.numVertices + 1
        newVertex = Vertex(key)
        self.vertList[key] = newVertex
        return newVertex

    def getVertex(self,n):
        if n in self.vertList:
            return self.vertList[n]
        else:
            return None

    def __contains__(self,n):
        return n in self.vertList

    def addEdge(self,f,t,cost=0):
        if f not in self.vertList:
            nv = self.addVertex(f)
        if t not in self.vertList:
            nv = self.addVertex(t)
        self.vertList[f].addNeighbor(self.vertList[t], cost)

    def getVertices(self):
        return self.vertList.keys()

    def __iter__(self):
        return iter(self.vertList.values())
```

*Listing 2*

使用刚才定义的 `Graph` 和 `Vertex` 类，下面的 Python 会话创建 Figure 2中的图。首先我们创建 6 个编号为 0 到 5 的顶点。然后我们展示顶点字典。 注意，对于每个键 0 到 5，我们创建了一个顶点的实例。接下来，我们添加将顶点连接在一起的边。 最后，嵌套循环验证图中的每个边缘是否正确存储。 你应该在本会话结束时根据 Figure 2检查边列表的输出是否正确。

```
>>> g = Graph()
>>> for i in range(6):
...    g.addVertex(i)
>>> g.vertList
{0: <adjGraph.Vertex instance at 0x41e18>,
 1: <adjGraph.Vertex instance at 0x7f2b0>,
 2: <adjGraph.Vertex instance at 0x7f288>,
 3: <adjGraph.Vertex instance at 0x7f350>,
 4: <adjGraph.Vertex instance at 0x7f328>,
 5: <adjGraph.Vertex instance at 0x7f300>}
>>> g.addEdge(0,1,5)
>>> g.addEdge(0,5,2)
>>> g.addEdge(1,2,4)
>>> g.addEdge(2,3,9)
>>> g.addEdge(3,4,7)
>>> g.addEdge(3,5,3)
>>> g.addEdge(4,0,1)
>>> g.addEdge(5,4,8)
>>> g.addEdge(5,2,1)
>>> for v in g:
...    for w in v.getConnections():
...        print("( %s , %s )" % (v.getId(), w.getId()))
...
( 0 , 5 )
( 0 , 1 )
( 1 , 2 )
( 2 , 3 )
( 3 , 4 )
( 3 , 5 )
( 4 , 0 )
( 5 , 4 )
( 5 , 2 )
```

*Figure 2*

# 7.7.字梯的问题

让我们从下面的叫字梯的难题开始图算法研究。将单词 “FOOL” 转换为单词 “SAGE”。 在字梯中你通过改变一个字母逐渐发生变化。 在每一步，你必须将一个字变换成另一个字。 字梯益智游戏是刘易斯卡罗尔 1878 年发明的，爱丽丝梦游仙境的作者。下面的单词序列示出了对上述问题的一种可能的解决方案。

```
FOOL
POOL
POLL
POLE
PALE
SALE
SAGE
```

有许多关于字梯问题的变种。例如，可能附加了完成转换的特定数量的步骤，或者可能需要使用特定的词。在本节中，我们将计算起始字转换为结束字所需的最小转换次数。

毫不奇怪，因为这一章是图，我们可以使用图算法解决这个问题。 这里是我们需要的步骤：

- 将字之间的关系表示为图。
- 使用称为广度优先搜索的图算法来找到从起始字到结束字的有效路径。

# 7.8.构建字梯图

我们的第一个问题是弄清楚如何将大量的单词集合转换为图。 如果两个词只有一个字母不同，我们就创建从一个词到另一个词的边。如果我们可以创建这样的图，则从一个词到另一个词的任意路径就是词梯子拼图的解决方案。 Figure 1展示了一些解决 `FOOL` 到 `SAGE` 字梯问题的单词的小图。 请注意，图是无向图，边未加权。

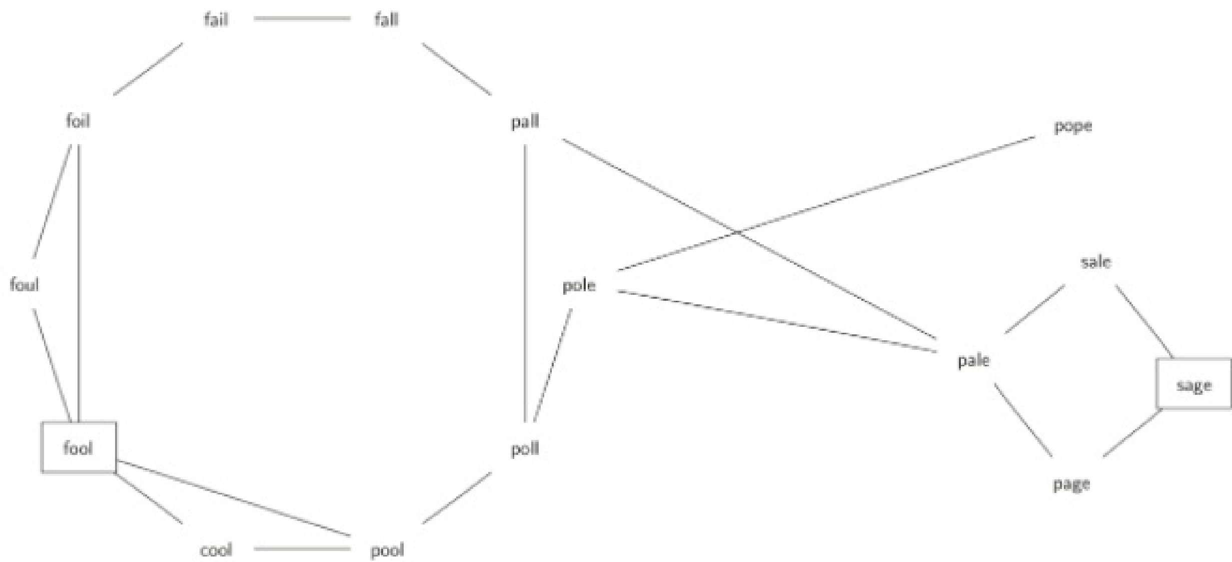



*Figure 1*

我们可以使用几种不同的方法来创建解决这个问题的图。假设我们有一个长度相同的单词列表。作为起点，我们可以在图中为列表中的每个单词创建一个顶点。为了弄清楚如何连接单词，我们可以比较列表中的每个单词。比较时我们看有多少字母是不同的。如果所讨论的两个字只有一个字母不同，我们可以在图中创建它们之间的边。对于小的列表，这种方法会正常工作;然而假设我们有一个 `5,110` 词的列表。粗略地说，将一个字与列表上的每个其他词进行比较是 O(n^2 )。对于5110 个词，n^2 是超过2600万的比较。

我们可以通过以下方法做得更好。假设我们有大量的桶，每个桶在外面有一个四个字母的单词，除了标签中的一个字母已经被下划线替代。例如，看 Figure 2，我们可能有一个标记为 “pop” 的桶。当我们处理列表中的每个单词时，我们使用 “” 作为通配符比较每个桶的单词，所以 “pope” 和 “pops “ 将匹配 ”pop_“。每次我们找到一个匹配的桶，我们就把单词放在那个桶。一旦我们把所有单词放到适当的桶中，就知道桶中的所有单词必须连接。

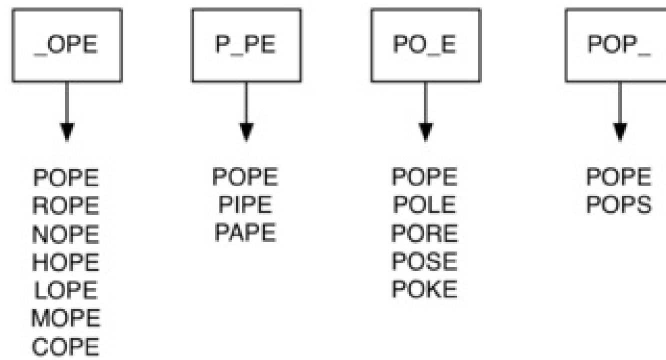



*Figure 2*

在 Python 中，我们使用字典来实现我们刚才描述的方案。我们刚才描述的桶上的标签是我们字典中的键。该键存储的值是单词列表。 一旦我们建立了字典，我们可以创建图。 我们通过为图中的每个单词创建一个顶点来开始图。 然后，我们在字典中的相同键下找到的所有顶点创建边。 Listing 1 展示了构建图所需的 Python 代码。

```
from pythonds.graphs import Graph

def buildGraph(wordFile):
    d = {}
    g = Graph()
    wfile = open(wordFile,'r')
    # create buckets of words that differ by one letter
    for line in wfile:
        word = line[:-1]
        for i in range(len(word)):
            bucket = word[:i] + '_' + word[i+1:]
            if bucket in d:
                d[bucket].append(word)
            else:
                d[bucket] = [word]
    # add vertices and edges for words in the same bucket
    for bucket in d.keys():
        for word1 in d[bucket]:
            for word2 in d[bucket]:
                if word1 != word2:
                    g.addEdge(word1,word2)
    return g
```

*Listing 1*

因为这是我们的第一个真实世界图问题，你可能想知道图是如何稀疏？这个问题的四个字母的单词列表是 `5,110` 字长。 如果我们使用邻接矩阵，则矩阵将具有 `5,110 * 5,110 = 26,112,100` 个格。 由 `buildGraph` 函数构造的图正好有 `53,286` 个边，所以矩阵只有 0.20%

的单元格填充！ 这是一个非常稀疏的矩阵。

# 7.9.实现广度优先搜索

通过构建图，我们现在可以将注意力转向我们将使用的算法来找到字梯问题的最短解。我们将使用的图算法称为“宽度优先搜索”算法。宽度优先搜索（BFS）是用于搜索图的最简单的算法之一。它也作为几个其他重要的图算法的原型，我们将在以后研究。

给定图 G 和起始顶点 s，广度优先搜索通过探索图中的边以找到 G 中的所有顶点，其中存在从 s 开始的路径。通过广度优先搜索，它找到和 s 相距 k 的所有顶点，然后找到距离为 k + 1 的所有顶点。可视化广度优先搜索算法一个好方法是想象它正在建一棵树，一次建一层。广度优先搜索先从其他起始顶点开始添加它的所有子节点，然后再添加其子节点的子节点。

为了跟踪进度，BFS 将每个顶点着色为白色，灰色或黑色。当它们被构造时，所有顶点被初始化为白色。白色顶点是未发现的顶点。当一个顶点最初被发现时它变成灰色的，当 BFS 完全探索完一个顶点时，它被着色为黑色。这意味着一旦顶点变黑色，就没有与它相邻的白色顶点。另一方面，灰色节点可能有与其相邻的一些白色顶点，表示仍有额外的顶点要探索。

下面 Listing 2 中所示的广度优先搜索算法使用我们先前开发的邻接表表示。此外，它使用一个 Queue，一个关键的地方，决定下一个探索的顶点。

此外，BFS 算法使用 Vertex 类的扩展版本。这个新的顶点类添加了三个新的实例变量： `distance` ， `predecessor` 和 `color` 。这些实例变量中的每一个还具有适当的 `getter` 和 `setter` 方法。这个扩展的顶点类代码包含在 `pythonds` 包中，但我们不会在这里展示它，因为没有新的需要学习的点。

BFS 从起始顶点开始，颜色从灰色开始，表明它正在被探索。另外两个值，即距离和前导，对于起始顶点分别初始化为 0 和 None 。最后，放到一个队列中。下一步是开始系统地检查队列前面的顶点。我们通过迭代它的邻接表来探索队列前面的每个新节点。当检查邻接表上的每个节点时，检查其颜色。如果它是白色的，顶点是未开发的，有四件事情发生：

1. 新的，未开发的顶点 nbr，被着色为灰色。
2. nbr 的前导被设置为当前节点 `currentVert`
3. 到 nbr 的距离设置为到 `currentVert + 1` 的距离
4. nbr 被添加到队列的末尾。 将 nbr 添加到队列的末尾有效地调度此节点以进行进一步探索，但不是直到 `currentVert` 的邻接表上的所有其他顶点都被探索。

```
from pythonds.graphs import Graph, Vertex
from pythonds.basic import Queue

def bfs(g,start):
  start.setDistance(0)
  start.setPred(None)
  vertQueue = Queue()
  vertQueue.enqueue(start)
  while (vertQueue.size() > 0):
    currentVert = vertQueue.dequeue()
    for nbr in currentVert.getConnections():
      if (nbr.getColor() == 'white'):
        nbr.setColor('gray')
        nbr.setDistance(currentVert.getDistance() + 1)
        nbr.setPred(currentVert)
        vertQueue.enqueue(nbr)
    currentVert.setColor('black')
```

*Listing 2*

让我们看看 bfs 函数如何构造对应于 Figure 1 中的图的广度优先树。开始我们取所有与 `fool` 相邻的节点，并将它们添加到树中。 相邻节点包括 `pool` , `foil` , `foul` , `cool` 。 这些节点被添加到新节点的队列以进行扩展。 Figure 3 展示了在此步骤之后树以及队列的状态。

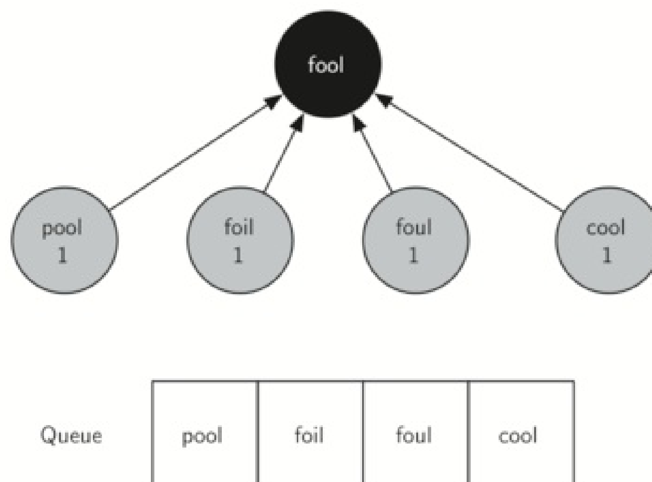



*Figure 3*

在下一步骤中，bfs 从队列的前面删除下一个节点（ `pool` ），并对其所有相邻节点重复该过程。 然而，当 bfs 检查节点 `cool` 时，它发现 `cool` 的颜色已经改变为灰色。这表明有一条较短的路径到 `cool` ，并且 `cool` 已经在队列上进一步扩展。在检查 `pool` 期间添加到队列的唯一新节点是 `poll` 。 树和队列的新状态如 Figure 4所示。

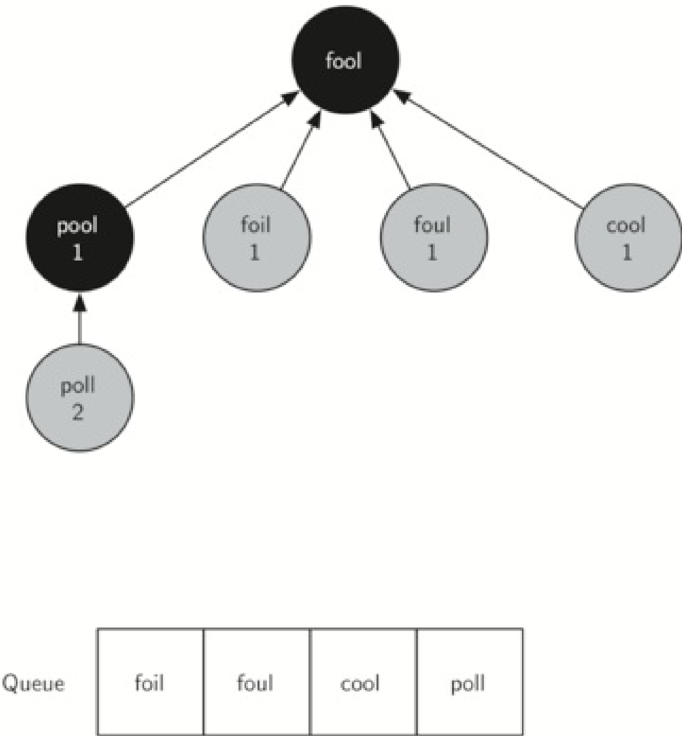



*Figure 4*

队列上的下一个顶点是 `foil` 。 `foil` 可以添加到树中的唯一新节点是 `fail` 。 当 bfs 继续处理队列时，接下来的两个节点都不向队列或树添加新内容。 Figure 5 展示了在树的第二级上展开所有顶点之后的树和队列。

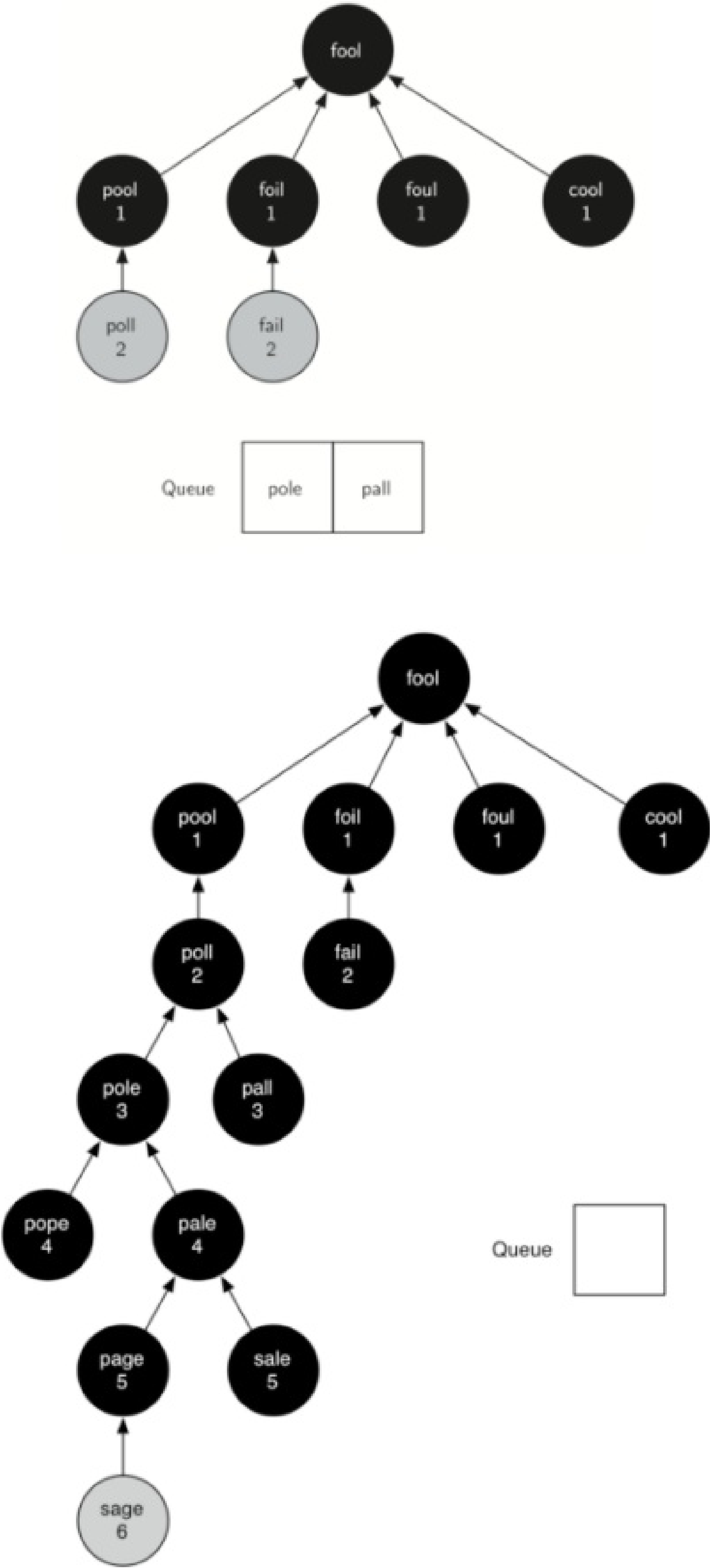

fool
pool
1
foil
1
foul
1
cool
1
poll
2
fail
2
Queue
pole
pall
fool
pool
1
foil
1
foul
1
cool
1
poll
2
fail
2
pole
3
pall
3
pope
4
pale
4
page
5
sale
5
sage
6
Queue

*Figure 5-6*

你应该自己继续完成算法，以便能够熟练使用它。Figure 6 展示了在 Figure 3 中的所有顶点都被扩展之后的最终广度优先搜索树。关于广度优先搜索解决方案的令人惊讶的事情是，我们不仅解决了我们开始的 `FOOL-SAGE` 问题，还解决了许多其他问题。 我们可以从广度优先搜索树中的任何顶点开始，并沿着前导箭头回到根，找到从任何字回到 `fool` 的最短的词梯。下面的函数（Listing 3）展示了如何按前导链接打印出字梯。

```
def traverse(y):
    x = y
    while (x.getPred()):
        print(x.getId())
        x = x.getPred()
    print(x.getId())

traverse(g.getVertex('sage'))
```

*Listing 3*

# 7.10.广度优先搜索分析

在继续使用其他图算法之前，让我们分析广度优先搜索算法的运行时性能。首先要观察的是，对于图中的每个顶点 `|V|` 最多执行一次 while 循环。因为一个顶点必须是白色，才能被检查和添加到队列。这给出了用于 while 循环的 O(v)。嵌套在 while 内部的 for 循环对于图中的每个边执行最多一次， `|E|` 。原因是每个顶点最多被出列一次，并且仅当节点 u 出队时，我们才检查从节点 u 到节点 v 的边。这给出了用于 for 循环的 O(E) 。组合这两个环路给出了 O(V+E)。

当然做广度优先搜索只是任务的一部分。从起始节点到目标节点的链接之后是任务的另一部分。最糟糕的情况是，如果图是单个长链。在这种情况下，遍历所有顶点将是 O(V)。正常情况将是 |V| 的一小部分但我们仍然写 O(V)。

最后，至少对于这个问题，存在构建初始图形所需的时间。我们把 `buildGraph` 函数的分析作为一个练习。

# 7.11.骑士之旅

另一个经典问题，我们可以用来说明第二个通用图算法称为“骑士之旅”。骑士之旅图是在一个棋盘上用一个棋子当骑士玩。图的目的是找到一系列的动作，让骑士访问板上的每格一次。一个这样的序列被称为“旅游”。骑士的旅游难题已经吸引了象棋玩家，数学家和计算机科学家多年。一个 8×8 棋盘的可能的游览次数的上限为 1.305×10^35 ;然而，还有更多可能的死胡同。显然，这是一个需要脑力，计算能力，或两者都需要的问题。

虽然研究人员已经研究了许多不同的算法来解决骑士的旅游问题，图搜索是最容易理解的程序之一。再次，我们将使用两个主要步骤解决问题：

- 表示骑士在棋盘上作为图的动作。
- 使用图算法来查找长度为 `rows×columns-1` 的路径，其中图上的每个顶点都被访问一次。

# 7.12.构建骑士之旅图

为了将骑士的旅游问题表示为图，我们将使用以下两个点：棋盘上的每个正方形可以表示为图形中的一个节点。骑士的每个合法移动可以表示为图形中的边。Figure 1 展示了骑士的移动以及图中的对应边。

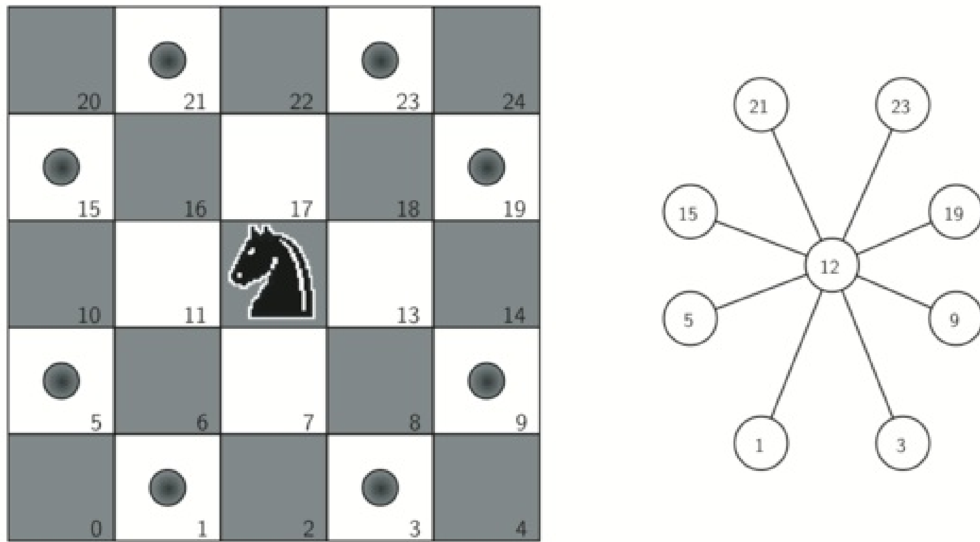



*Figure 1*

要构建一个 `n*n` 的完整图，我们可以使用 Listing 1 中所示的 Python 函数。`knightGraph` 函数在整个板上进行一次遍历。在板上的每个方块上，`knightGraph` 函数调用 `genLegalMoves`，为板上的位置创建一个移动列表。所有移动在图形中转换为边。另一个帮助函数 `posToNodeId` 按照行和列将板上的位置转换为类似于 Figure 1 所示的顶点数的线性顶点数。

```
from pythonds.graphs import Graph

def knightGraph(bdSize):
    ktGraph = Graph()
    for row in range(bdSize):
       for col in range(bdSize):
           nodeId = posToNodeId(row,col,bdSize)
           newPositions = genLegalMoves(row,col,bdSize)
           for e in newPositions:
               nid = posToNodeId(e[0],e[1],bdSize)
               ktGraph.addEdge(nodeId,nid)
    return ktGraph

def posToNodeId(row, column, board_size):
    return (row * board_size) + column
```

*Listing 1*

`genLegalMoves` 函数（Listing 2）使用板上骑士的位置，并生成八个可能移动中的一个。`legalCoord` 辅助函数（Listing 2）确保生成的特定移动仍在板上。

```
def genLegalMoves(x,y,bdSize):
    newMoves = []
    moveOffsets = [(-1,-2),(-1,2),(-2,-1),(-2,1),
                   ( 1,-2),( 1,2),( 2,-1),( 2,1)]
    for i in moveOffsets:
        newX = x + i[0]
        newY = y + i[1]
        if legalCoord(newX,bdSize) and \
                        legalCoord(newY,bdSize):
            newMoves.append((newX,newY))
    return newMoves

def legalCoord(x,bdSize):
    if x >= 0 and x < bdSize:
        return True
    else:
        return False
```

*Listing 2*

Figure 2 展示了一个 8×8 板的可能移动的完整图。图中有正好 336 个边。 注意，与板的边相对应的顶点具有比板中间的顶点更少的连接（移动数）。 再次我们可以看到图的稀疏。 如果图形完全连接，则会有 4,096 个边。 由于只有336 个边，邻接矩阵只有 8.2% 填充率。

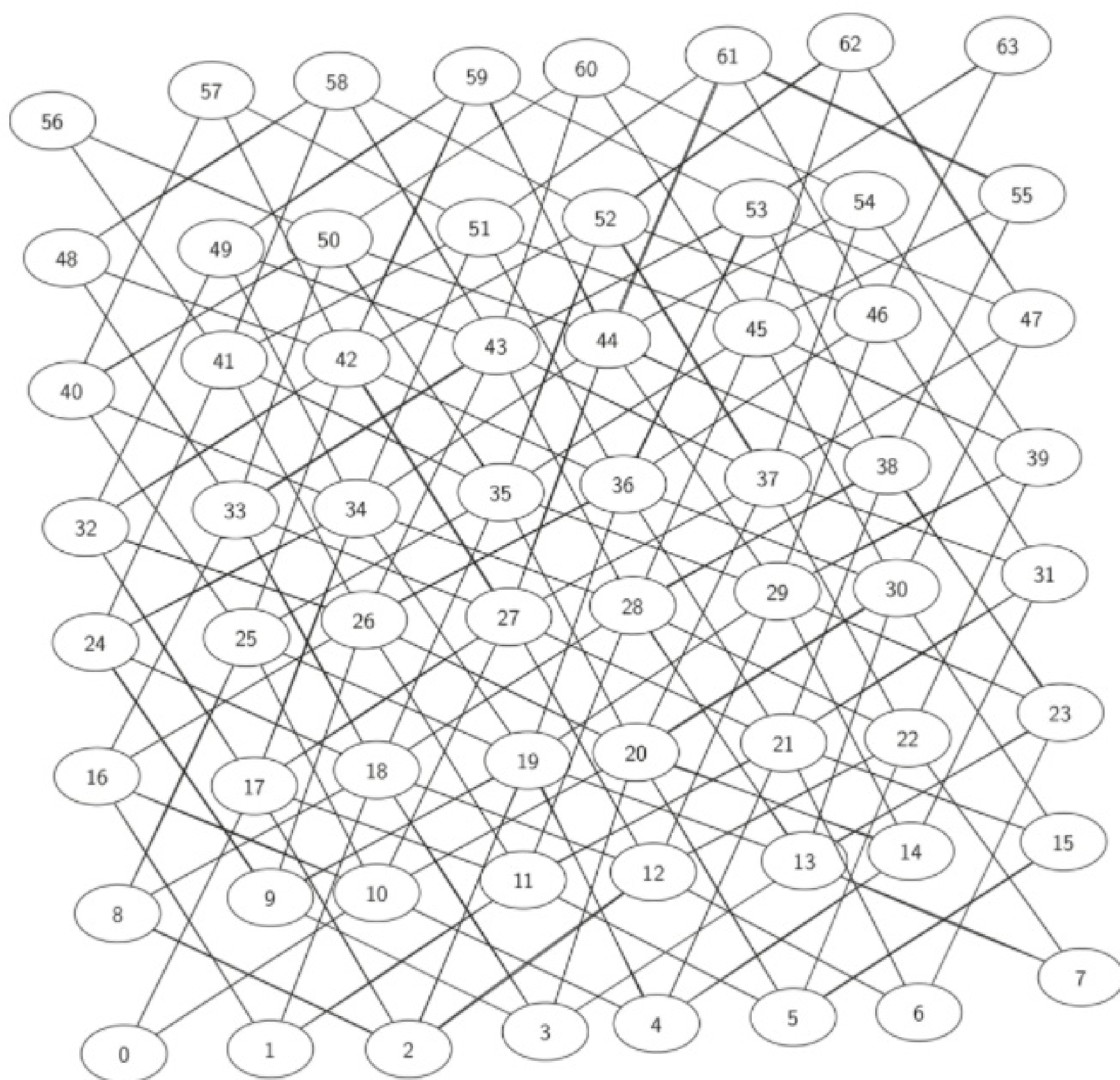

56
57
58
59
60
61
62
63
48
49
50
51
52
53
54
55
40
41
42
43
44
45
46
47
32
33
34
35
36
37
38
39
24
25
26
27
28
29
30
31
16
17
18
19
20
21
22
23
8
9
10
11
12
13
14
15
0
1
2
3
4
5
6
7

*Figure 2*

# 7.13.实现骑士之旅

我们将用来解决骑士旅游问题的搜索算法称为 深度优先搜索（DFS）。尽管在前面部分中讨论的广度优先搜索算法一次建立一个搜索树，但是深度优先搜索通过尽可能深地探索树的一个分支来创建搜索树。在本节中，我们将介绍实现深度优先搜索的两种算法。我们将看到的第一个算法通过明确地禁止一个节点被访问多次来直接解决骑士的旅行问题。第二种实现是更通用的，但是允许在构建树时多次访问节点。第二个版本在后续部分中用于开发其他图形算法。

图的深度优先搜索正是我们需要的，来找到有 63 个边的路径。我们将看到，当深度优先搜索算法找到死角（图中没有可移动的地方）时，它将回到下一个最深的顶点，允许它进行移动。

`knightTour` 函数有四个参数：`n` ，搜索树中的当前深度; `path` ，到此为止访问的顶点的列表; `u` ，图中我们希望探索的顶点; `limit` 路径中的节点数。 `knightTour` 函数是递归的。当调用 `knightTour` 函数时，它首先检查基本情况。如果我们有一个包含 `64` 个顶点的路径，我们状态为 `True` 的 `knightTour` 返回，表示我们找到了一个成功的线路。如果路径不够长，我们继续通过选择一个新的顶点来探索一层，并对这个顶点递归调用knightTour。

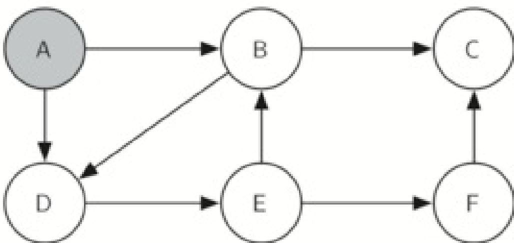
DFS 还使用颜色来跟踪图中的哪些顶点已被访问。未访问的顶点是白色的，访问的顶点是灰色的。如果已经探索了特定顶点的所有邻居，并且我们尚未达到64个顶点的目标长度，我们已经到达死胡同。当我们到达死胡同时，我们必须回溯。当我们从状态为 False 的 knightTour 返回时，发生回溯。在广度优先搜索中，我们使用一个队列来跟踪下一个要访问的顶点。由于深度优先搜索是递归的，我们隐式使用一个栈来帮助我们回溯。当我们从第 11 行的状态为 False 的 `knightTour` 调用返回时，我们保持在 while 循环中，并查看 `nbrList` 中的下一个顶点。

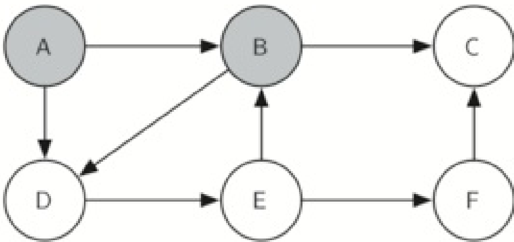
```
from pythonds.graphs import Graph, Vertex
def knightTour(n,path,u,limit):
        u.setColor('gray')
        path.append(u)
        if n < limit:
            nbrList = list(u.getConnections())
            i = 0
            done = False
            while i < len(nbrList) and not done:
                if nbrList[i].getColor() == 'white':
                    done = knightTour(n+1, path, nbrList[i], limit)
                i = i + 1
            if not done:  # prepare to backtrack
                path.pop()
                u.setColor('white')
        else:
            done = True
        return done
```

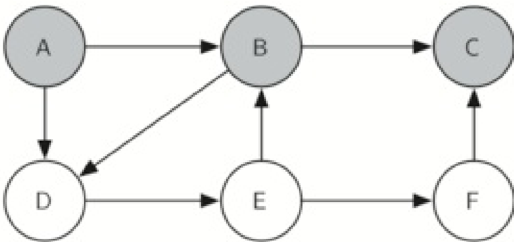
*Listing 3*

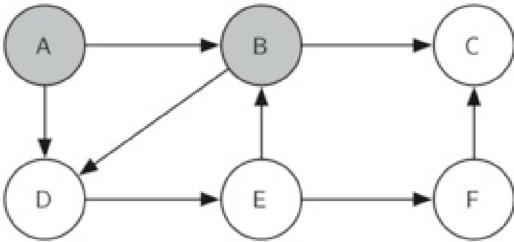
让我们看看一个简单的例子 `knightTour` 。你可以按照搜索的步骤参考下面的图。对于这个例子，我们假设对第 6 行的 `getConnections` 方法的调用按字母顺序对节点排序。我们首先调用 `knightTour(0，path，A，6)`

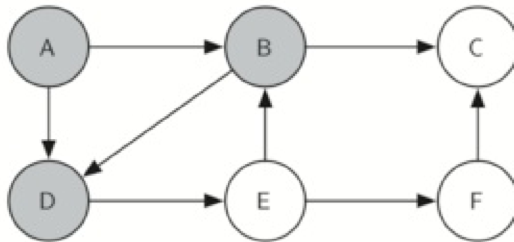
Figure 中 `knightTour` 从节点 A 开始.与 A 相邻的节点是 B 和 D。由于 B 在字母 D 之前，DFS选择 B 展开下一个，如 Figure 4 所示。当 `knightTour` 被递归调用时，开始从 B 开始探寻。 B 与 C 和 D 相邻，所以 knightTour 选择接下来探索 C。然而，如 Figure 5 所示，节点 C 是没有相邻节点的死胡同。此时，我们将节点 C 的颜色更改为白色。对 knightTour 的调用返回值 False。从递归调用的返回有效地将搜索回溯到顶点B（参见Figure 6）。列表中要探索的下一个顶点是顶点 D，因此 `knightTour` 使递归调用移动到节点 D（参见 Figure 7）。从顶点 D 开始， `knightTour` 可以继续进行递归调用，直到我们再次到达节点 C（参见Figure 8，Figure 9和 Figure 10）。然而，当我们到达节点C时，测试 `n <limit` 失败，所以我们知道已经耗尽了图中的所有节点。在这一点上，我们可以返回 True，表示我们已经成功地浏览了图。当我们返回列表时，路径具有值[A，B，D，E，F，C]，这是我们需要遍历图以访问每个节点的顺序。


A
B
C
D
E
F


A
B
C
D
E
F


A
B
C
D
E
F


A
B
C
D
E
F


A
B
C
D
E
F


A
B
C

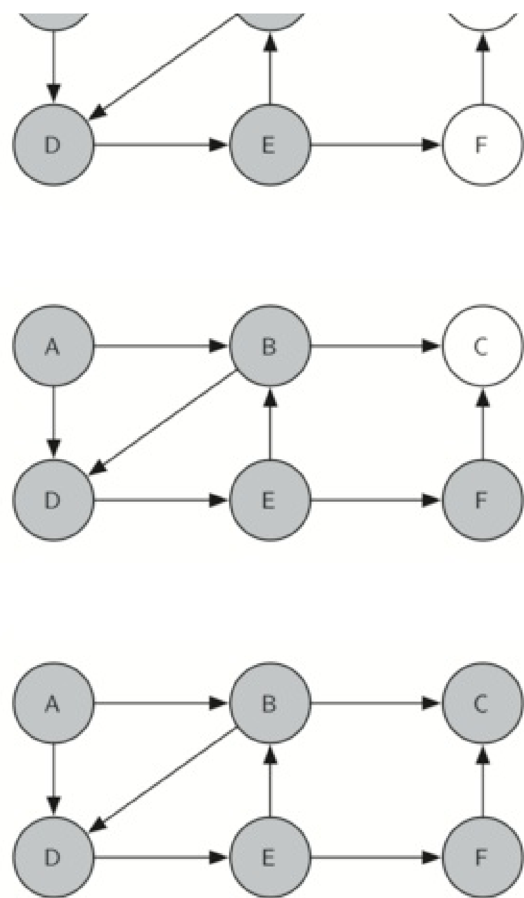



*Figure 3-10*

Figure 11 展示了一个 8×8 板的完整遍历。有许多可能的路径; 一些是对称的。 通过一些修改，你可以使遍历开始和结束在同一个正方形。

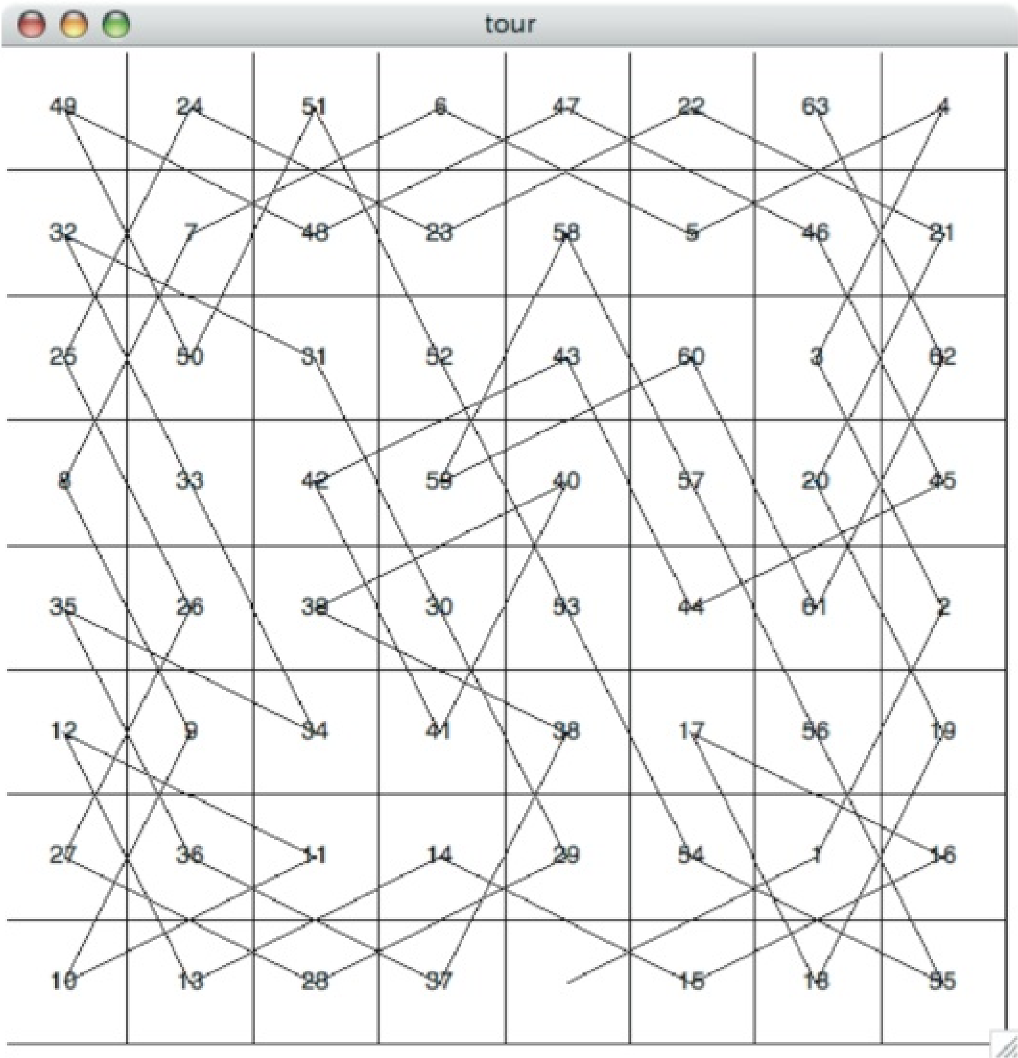
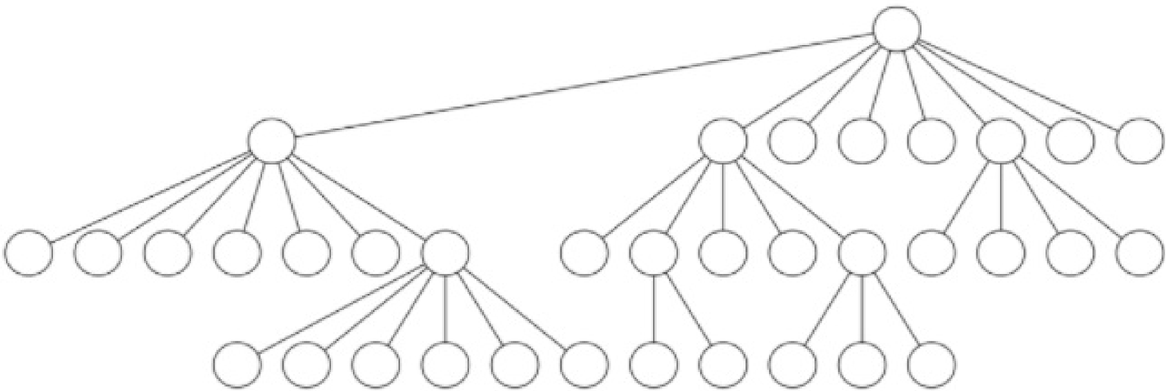

tour
49 24 51 6 47 22 63 4
32 7 48 23 58 5 46 21
25 50 31 52 43 60 3 62
8 33 42 59 40 57 20 45
35 26 39 30 53 44 61 2
12 9 34 41 38 17 56 19
27 36 11 14 29 54 1 16
10 13 28 37 15 18 55

*Figure 10*

# 7.14.骑士之旅分析

有最后关于骑士之旅一个有趣的话题，然后我们将继续到深度优先搜索的通用版本。主题是性能。特别是， `knightTour` 对于你选择下一个要访问的顶点的方法非常敏感。例如，在一个5乘5的板上，你可以在快速计算机上处理路径花费大约1.5秒。但是如果你尝试一个 8×8 的板，会发生什么？在这种情况下，根据计算机的速度，你可能需要等待半小时才能获得结果！这样做的原因是我们到目前为止所实现的骑士之旅问题是大小为 O(k^N ) 的指数算法，其中 N 是棋盘上的方格数，k 是小常数。Figure 12 可以帮助我们搞清楚为什么会这样。树的根表示搜索的起点。从那里，算法生成并检查骑士可以做出的每个可能的移动。正如我们之前注意到的，可能的移动次数取决于骑士在板上的位置。在角落只有两个合法的动作，在角落邻近的正方形有三个，在板的中间有八个。Figure 13 展示了板上每个位置可能的移动次数。在树的下一级，再次有 2 到 8 个可能的下一个移动。要检查的可能位置的数量对应于搜索树中的节点的数量。



| 2 | 3 | 4 | 4 | 4 | 4 | 3 | 2 |
|---|---|---|---|---|---|---|---|
| 3 | 4 | 6 | 6 | 6 | 6 | 4 | 3 |
| 4 | 6 | 8 | 8 | 8 | 8 | 6 | 4 |
| 4 | 6 | 8 | 8 | 8 | 8 | 6 | 4 |
| 4 | 6 | 8 | 8 | 8 | 8 | 6 | 4 |
| 4 | 6 | 8 | 8 | 8 | 8 | 6 | 4 |
| 3 | 4 | 6 | 6 | 6 | 6 | 4 | 3 |
| 2 | 3 | 4 | 4 | 4 | 4 | 3 | 2 |

*Figure 12-13*

我们已经看到，高度 N 的二叉树中的节点数量是 2^N+1 - 1。对于具有可以具有多达八个孩子而不是两个节点的树，节点的数量要大得多。因为每个节点的分支因子是可变的，我们可以使用平均分支因子估计节点的数量。重要的是要注意，这个算法是指数：k^N+1 - 1，其中 k 是板的平均分支因子。让我们看看这增长有多快！对于 5×5 的板，树将是 25 级深，或者 N = 24，将第一级算为级 0。平均分支因子是 k = 3.8 因此，搜索树中的节点数量是 3.8^25 - 1 或 3.12×10^14 。对于 6x6 板，k = 4.4，有 1.5×10^23 个节点，对于常规的 8x8 棋盘，k = 5.25 ，有 1.3×10^46 。当然，由于问题有多个解决方案，我们不必去探索每个节点，但是我们必须探索的节点的小数部分只是一个不会改变问题的指数性质的常数乘数。我们将把它作为一个练习，看看你是否可以表示k 作为板的大小的函数。

幸运的是有一种方法来加速八乘八的情况，使其在一秒钟内运行完成。在下面的列表中，我们将展示加速 `knightTour` 的代码。这个函数（见Listing 4），被称为 `orderbyAvail` 将被用来代替上面代码中对 `u.getConnections` 的调用。 `orderByAvail` 函数中的关键是第 10 行。此行确保我们选择具有最少可用移动的下一个顶点。你可能认为这具有相反效果; 为什么不选择具有最多可用移动的节点？你可以通过运行该程序并在排序后插入行 `resList.reverse()` 来尝试该方法。

使用具有最多可用移动的顶点作为路径上的下一个顶点的问题是，它倾向于让骑士在游览中早访问中间的方格。当这种情况发生时，骑士很容易陷入板的一侧，在那里它不能到达在板的另一侧的未访问的方格。另一方面，访问具有最少可用移动的方块首先推动骑士访问围绕板的边缘的方块。这确保了骑士能够尽早地访问难以到达的角落，并且只有在必要时才使用中间的方块跳过棋盘。利用这种知识加速算法被称为启发式。人类每天都使用启发式来帮助做出决策，启发式搜索通常用于人工智能领域。这个特定的启发式称为 `Warnsdorff` 算法，由 `H. C. Warnsdorff` 命名，他在 1823 年发表了他的算法。

```
def orderByAvail(n):
    resList = []
    for v in n.getConnections():
        if v.getColor() == 'white':
            c = 0
            for w in v.getConnections():
                if w.getColor() == 'white':
                    c = c + 1
            resList.append((c,v))
    resList.sort(key=lambda x: x[0])
    return [y[1] for y in resList]
Next Section - 7.15. General Depth First Search
```

# 7.15.通用深度优先搜索

骑士之旅是深度优先搜索的特殊情况，其目的是创建最深的第一棵树，没有任何分支。更一般的深度优先搜索实际上更容易。它的目标是尽可能深的搜索，在图中连接尽可能多的节点，并在必要时创建分支。

甚至可能的是，深度优先搜索将创建多于一个树。当深度优先搜索算法创建一组树时，我们称之为深度优先森林。与广度优先搜索一样，我们的深度优先搜索使用前导链接来构造树。此外，深度优先搜索将在顶点类中使用两个附加的实例变量。新实例变量是发现和完成时间。发现时间跟踪首次遇到顶点之前的步骤数。完成时间是顶点着色为黑色之前的步骤数。正如我们看到的算法，节点的发现和完成时间提供了一些有趣的属性，我们可以在以后的算法中使用。

我们深度优先搜索的代码如 Listing 5 所示。由于 dfs 和它的辅助函数`dfsvisit` 这两个函数使用一个变量来跟踪调用 `dfsvisit` 的时间，所以我们选择将代码实现为继承自 `Graph` 类。此实现通过添加时间实例变量和两个方法 `dfs` 和 `dfsvisit`来扩展 `Graph` 类。看看第 11 行，你会注意到，`dfs` 方法在调用 `dfsvisit` 的图中所有的顶点迭代，这些节点是白色的。我们迭代所有节点而不是简单地从所选择的起始节点进行搜索的原因是为了确保图中的所有节点都被考虑到，没有顶点从深度优先森林中被遗漏。`for aVertex in self` 语句可能看起来不寻常，但请记住，在这种情况下，`self` 是 `DFSGraph` 类的一个实例，遍历实例中的所有顶点是一件自然的事情。

```
from pythonds.graphs import Graph
class DFSGraph(Graph):
    def __init__(self):
        super().__init__()
        self.time = 0

    def dfs(self):
        for aVertex in self:
            aVertex.setColor('white')
            aVertex.setPred(-1)
        for aVertex in self:
            if aVertex.getColor() == 'white':
                self.dfsvisit(aVertex)

    def dfsvisit(self,startVertex):
        startVertex.setColor('gray')
        self.time += 1
        startVertex.setDiscovery(self.time)
        for nextVertex in startVertex.getConnections():
            if nextVertex.getColor() == 'white':
                nextVertex.setPred(startVertex)
                self.dfsvisit(nextVertex)
        startVertex.setColor('black')
        self.time += 1
        startVertex.setFinish(self.time)
```

*Listing 5*

虽然我们 `bfs` 的实现只对有一条路径回到开始的路径的节点感兴趣，但是有可能创建一个宽度优先森林，其表示图中的所有节点之间的最短路径。我们把这作为一个练习。在接下来的两个算法中，我们将看到为什么跟踪深度优先森林的深度很重要。

`dfsvisit` 方法从名为 `startVertex` 的单个顶点开始，并尽可能深地探查所有相邻的白色顶点。如果仔细查看 `dfsvisit` 的代码并将其与广度优先搜索进行比较，应该注意的是，`dfsvisit` 算法几乎与 `bfs` 相同，除了在内部 for 循环的最后一行，`dfsvisit` 将自行递归调用以继续在更深的级别搜索，而 `bfs` 将节点添加到队列以供稍后探查。有趣的是，`bfs` 使用队列，`dfsvisit` 使用栈。你在代码中没有看到栈，但是它在 `dfsvisit` 的递归调用中是隐含的。

以下图的序列展示了针对小图的深度优先搜索算法。在这些图中，虚线指示检查的边，但是在边的另一端的节点已经被添加到深度优先树。在代码中，通过检查另一个节点的颜色是非白色的。

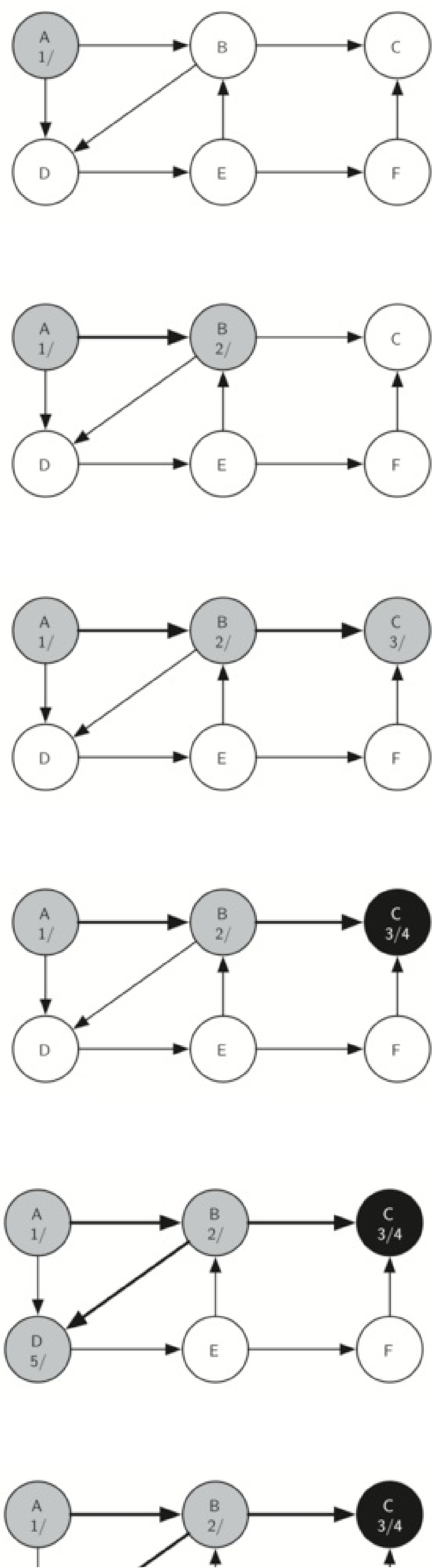
搜索从图的顶点 A 开始（Figure 14）。由于所有顶点在搜索开始时都是白色的，所以算法访问顶点 A。访问顶点的第一步是将颜色设置为灰色，这表示正在探索顶点，并且将发现时间设置为1，由于顶点 A 具有两个相邻的顶点（B，D），因此每个顶点也需要被访问。我们将做出任意决定，我们将按字母顺序访问相邻顶点。

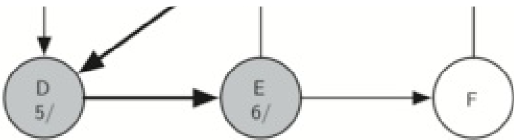
接下来访问顶点B（Figure 15），因此其颜色设置为灰色并且其发现时间被设置为 2。顶点 B 也与两个其他节点（C，D）相邻，因此我们将遵循字母顺序和访问节点 C 接下来。

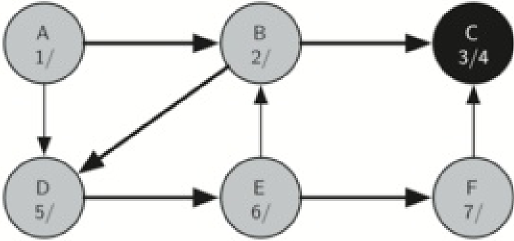
访问顶点 C（Figure16）使我们到树的一个分支的末端。在将节点灰色着色并将其发现时间设置为 3 之后，算法还确定没有与 C 相邻的顶点。这意味着我们完成了对节点 C 的探索，因此我们可以将顶点着色为黑色，并将完成时间设置为 4，在Figure 17 中，可以看到我们的搜索的状态。

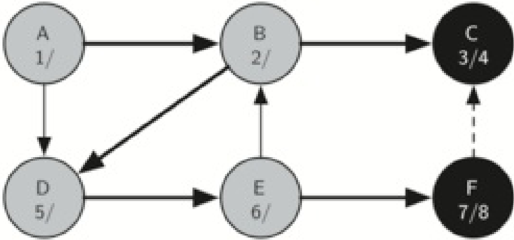
由于顶点 C 是一个分支的结束，我们现在返回到顶点 B，继续探索与 B 相邻的节点。从 B 中探索的唯一额外的顶点是 D，所以我们现在可以访问 D（Figure 18），并继续搜索顶点 D。顶点 D 快速引导我们到顶点 E（Figure 19）。顶点 E 具有两个相邻的顶点 B 和 F 。通常我们将按字母顺序探索这些相邻顶点，但是由于 B 已经是灰色的，所以算法识别出它不应该访问 B，因为这样做会将算法置于循环中！因此，继续探索列表中的下一个顶点，即 F（Figure 20）。

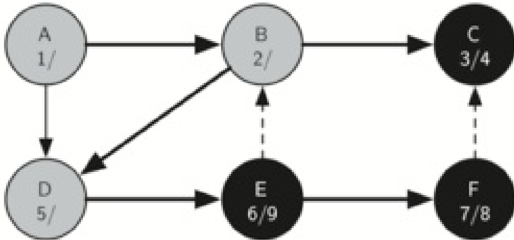
顶点 F 只有一个相邻的顶点 C，但由于 C 是黑色的，没有别的东西可以探索，算法已经到达另一个分支的结束。从这里开始，你将在 Figure 21至 Figure 25中看到算法运行回到第一个节点，设置完成时间和着色顶点为黑色。

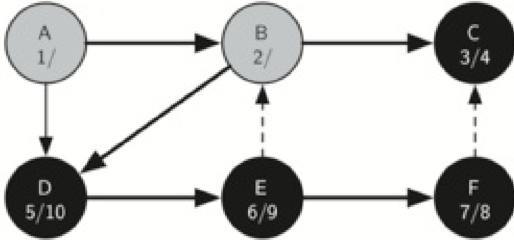

A
1/
B
C
D
E
F
A
1/
B
2/
C
D
E
F
A
1/
B
2/
C
3/
D
E
F
A
1/
B
2/
C
3/4
D
E
F
A
1/
B
2/
C
3/4
D
5/
E
F
A
1/
B
2/
C
3/4

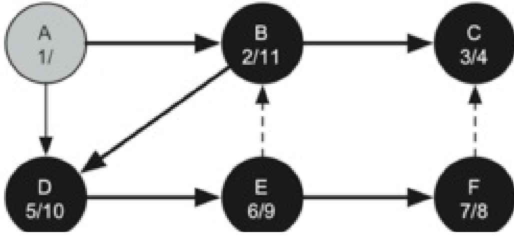

D
5/
E
6/
F


A
1/
B
2/
C
3/4
D
5/
E
6/
F
7/


A
1/
B
2/
C
3/4
D
5/
E
6/
F
7/8


A
1/
B
2/
C
3/4
D
5/
E
6/9
F
7/8


A
1/
B
2/
C
3/4
D
5/10
E
6/9
F
7/8


A
1/
B
2/11
C
3/4
D
5/10
E
6/9
F
7/8

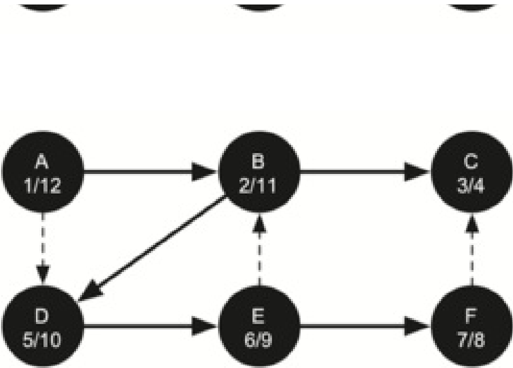



*Figure 14-25*

每个节点的开始和结束时间展示一个称为 `括号属性` 的属性。 该属性意味着深度优先树中的特定节点的所有子节点具有比它们的父节点更晚的发现时间和更早的完成时间。 Figure 26 展示了由深度优先搜索算法构造的树。

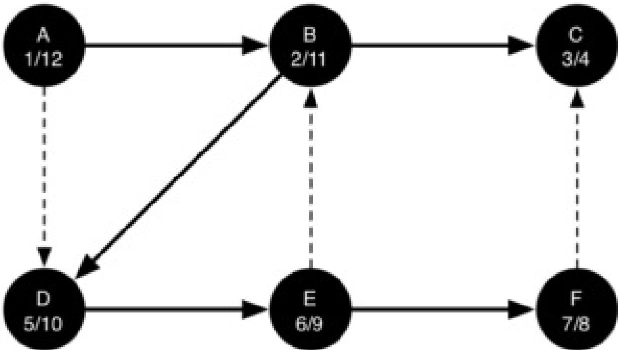



*Figure 26*

# 7.16.深度优先搜索分析

深度优先搜索的一般运行时间如下。 dfs 中的循环都在 O(V) 中运行，不计入 `dfsvisit` 中发生的情况，因为它们对图中的每个顶点执行一次。 在 `dfsvisit` 中，对当前顶点的邻接表中的每个边执行一次循环。 由于只有当顶点为白色时， `dfsvisit` 才被递归调用，所以循环对图中的每个边或 O(E) 执行最多一次。 因此，深度优先搜索的总时间是 O(V + E)。

# 7.17.拓扑排序

为了表明计算机科学家可以把任何东西变成一个图问题，让我们考虑做一批煎饼的问题。 菜谱真的很简单：1个鸡蛋，1杯煎饼粉，1汤匙油 和 3/4 杯牛奶。 要制作煎饼，你必须加热炉子，将所有的成分混合在一起，勺子搅拌。 当开始冒泡，你把它们翻过来，直到他们底部变金黄色。 在你吃煎饼之前，你会想要加热一些糖浆。 Figure 27将该过程示为图。

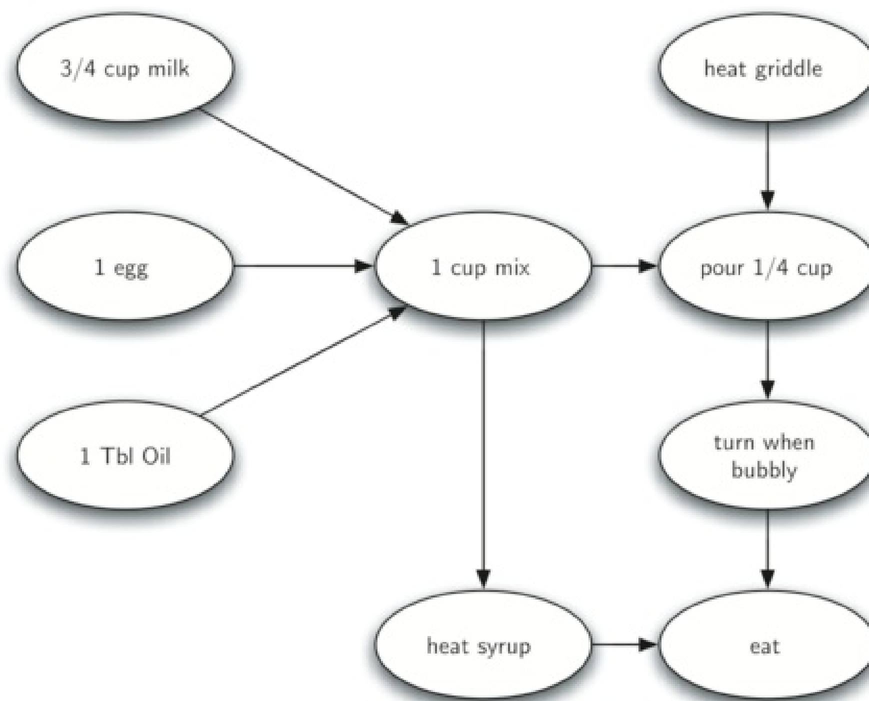



*Figure 27*

制作煎饼的困难是知道先做什么。从 Figure 27 可以看出，你可以从加热煎饼开始，或通过添加任何成分到煎饼。为了帮助我们决定应该做的每一个步骤的精确顺序，我们转向一个图算法称为 `拓扑排序` 。

拓扑排序采用有向无环图，并且产生所有其顶点的线性排序，使得如果图 G 包含边（v，w），则顶点 v 在排序中位于顶点 w 之前。定向非循环图在许多应用中使用以指示事件的优先级。制作煎饼只是一个例子;其他示例包括软件项目计划，用于数据库查询的优先图以及乘法矩阵。

拓扑排序是深度优先搜索的简单但有用的改造。拓扑排序的算法如下：

1. 对于某些图 g 调用 dfs(g)。我们想要调用深度优先搜索的主要原因是计算每个顶点的完成时间。
2. 以完成时间的递减顺序将顶点存储在列表中。

3. 返回有序列表作为拓扑排序的结果。

Figure 28 展示了在 Figure 26 所示的薄煎饼制作图上由 dfs 构建的深度优先森林。

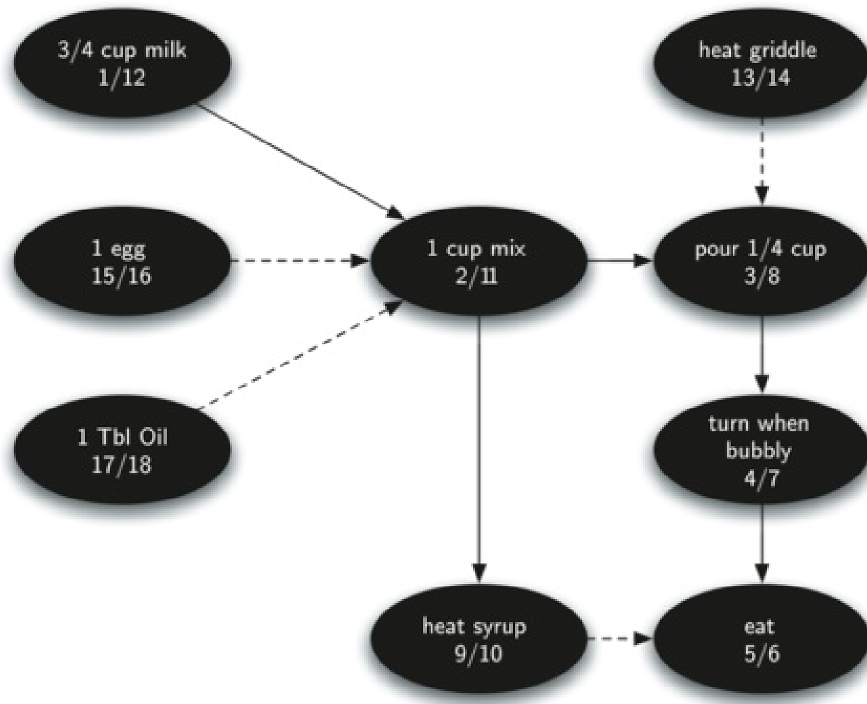



*Figure 28*

最后，Figure 29 展示了将拓扑排序算法应用于我们的图形的结果。 现在所有的分支已被删除，我们知道确切的做煎饼的步骤顺序。

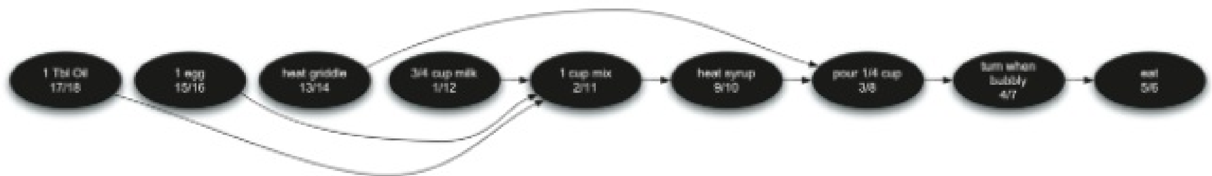



*Figure 29*

# 7.18.强连通分量

在本章的剩余部分，我们将把注意力转向一些非常大的图。我们将用来研究一些附加算法的图，由互联网上的主机之间的连接和网页之间的链接产生的图。 我们将从网页开始。

像 Google 和 Bing 这样的搜索引擎利用了网页上的页面形成非常大的有向图。 为了将万维网变换为图，我们将把一个页面视为一个顶点，并将页面上的超链接作为将一个顶点连接到另一个顶点的边缘。 Figure 30 展示了从 Luther College 的计算机科学主页开始，通过跟踪从一页到下一页的链接产生的图的非常小的部分。当然，这个图可能是巨大的，所以我们把它限制在距离 CS 主页不超过 10 个链接的网站。

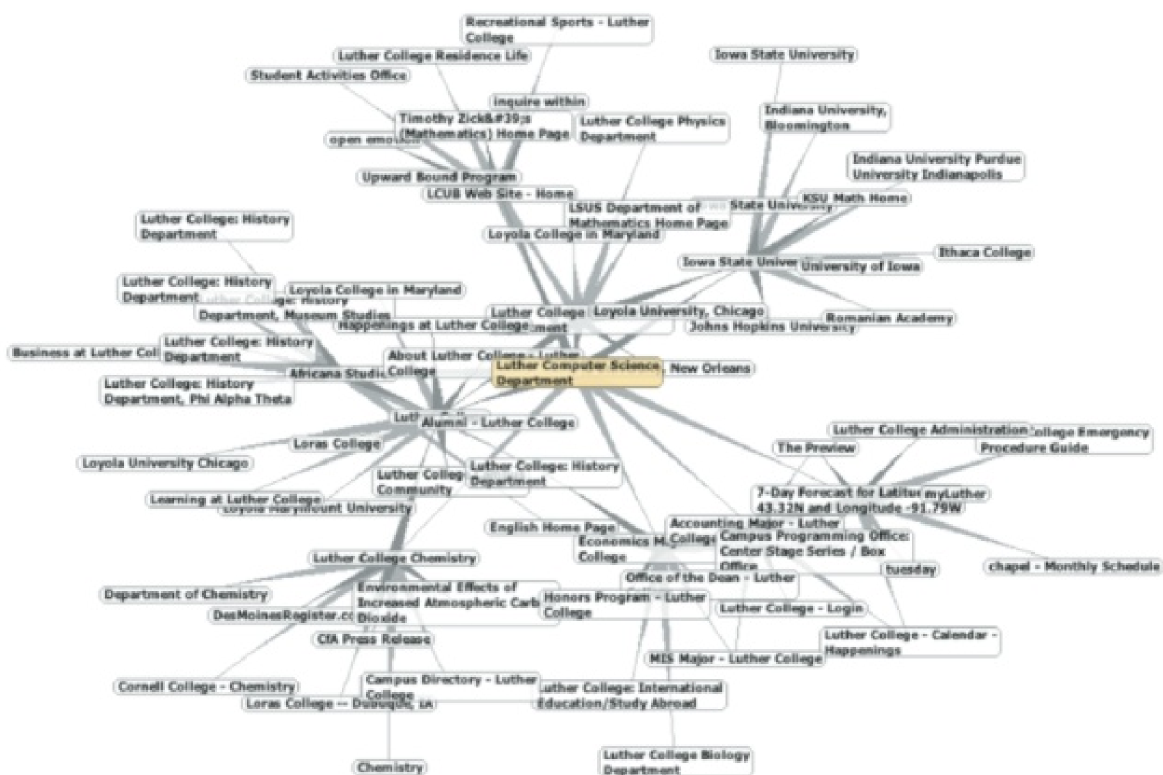



*Figure 30*

如果你看 Figure 30中的图形，你可能会有一些有趣的观察。首先你可能会注意到，图上的许多其他网站是其他路德学院网站。第二，你可能注意到有几个链接到爱荷华州的其他学院。第三，你可能注意到有几个链接到其他文理学院。你可能会得出这样的结论，网络集群上的网站在一些级别上底层结构类似。

可以帮助找到图中高度互连的顶点的集群的一种图算法被称为强连通分量算法（SCC）。我们正式定义图 G 的强连通分量 C 作为顶点 C⊂V 的最大子集，使得对于每对顶点 v,w∈C，我们具有从 v 到 w 的路径和从 w 到 v 的路径。Figure 27 展示了具有三个强连接分量的简单

图。强连接分量由不同的阴影区域标识。

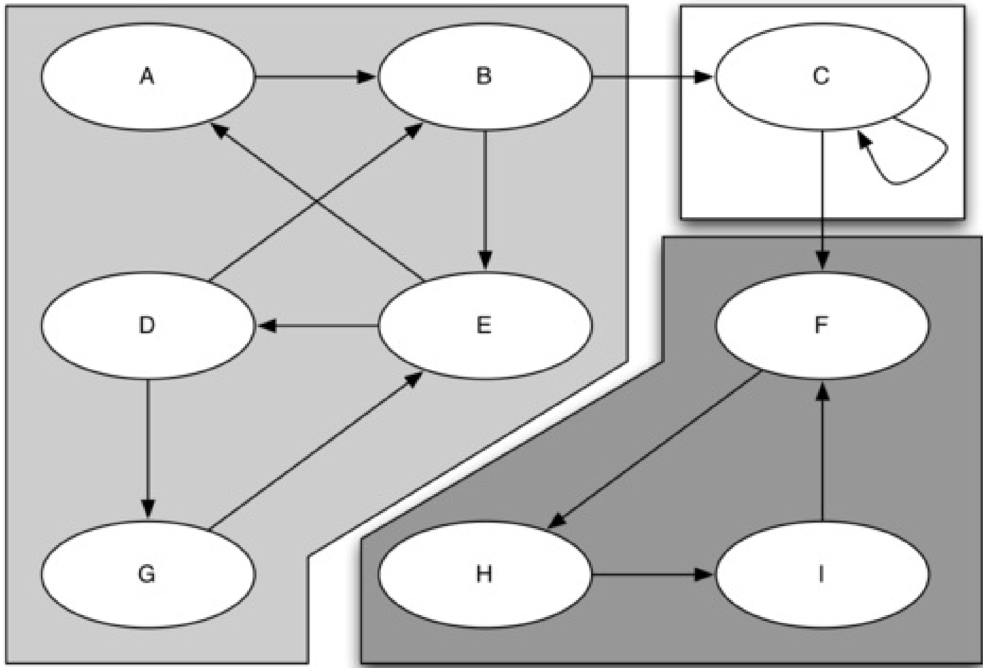



*Figure 27*

一旦确定了强连通分量，我们就可以通过将一个强连通分量中的所有顶点组合成一个较大的顶点来显示该图的简化视图。 Figure 31中的曲线图的简化版本如 Figure 32所示。

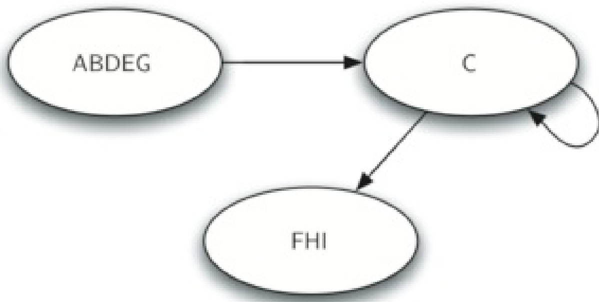



*Figure 32*

我们再次看到，我们可以通过使用深度优先搜索来创建一个非常强大和高效的算法。 在我们处理主 SCC 算法之前，我们必须考虑另一个定义。 图 G 的转置被定义为图 G^T ，其中图中的所有边已经反转。 也就是说，如果在原始图中存在从节点 A 到节点 B 的有向边，则 G^T 将包含从节点 B 到节点 A 的边。Figure 33和 Figure 34 展示了简单图及其变换。

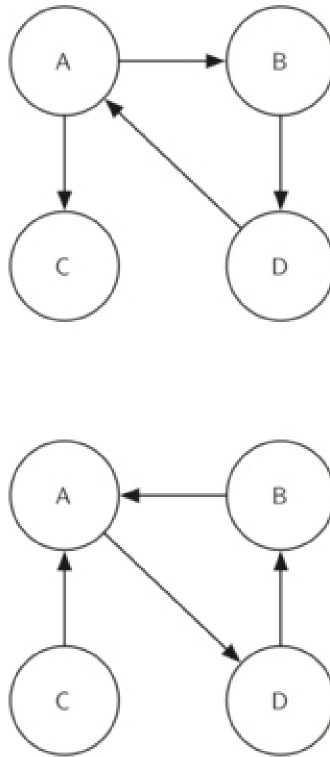


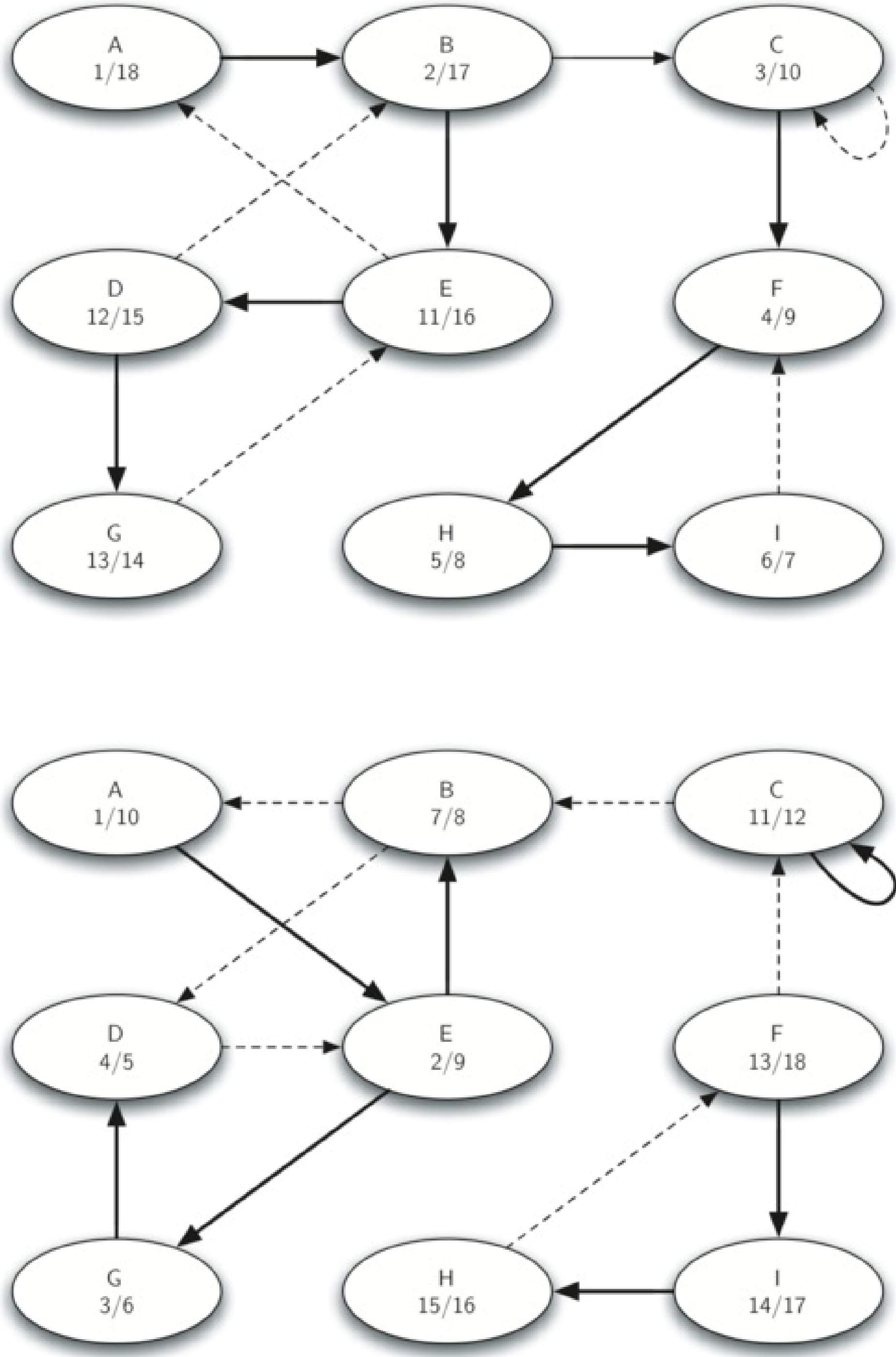

*Figure 33-34*

再看看数字。 请注意，Figure 33中的图形有两个强连通分量。 现在看看Figure 34。注意它也有两个强连通分量。

我们现在可以描述用于计算图的强连通分量的算法。

1. 调用 dfs 为图 G 计算每个顶点的完成时间。
2. 计算 G^T 。
3. 为图 G^T 调用 dfs，但在 DFS 的主循环中，以完成时间的递减顺序探查每个顶点。
4. 在步骤 3 中计算的森林中的每个树是强连通分量。输出森林中每个树中每个顶点的顶点标识组件。

让我们在 Figure 31中的示例图上跟踪上述步骤的操作。Figure 35 展示了由DFS 算法为原始图计算的开始和结束时间。 Figure 36 展示了通过在转置图上运行 DFS 计算的开始和结束时




间。

*Figure 36*

最后，Figure 37 展示了在强连通分量算法的步骤 3 中产生的三棵树的森林。 你会注意到，我们不为你提供 SCC 算法的 Python 代码，我们将此程序作为练习。

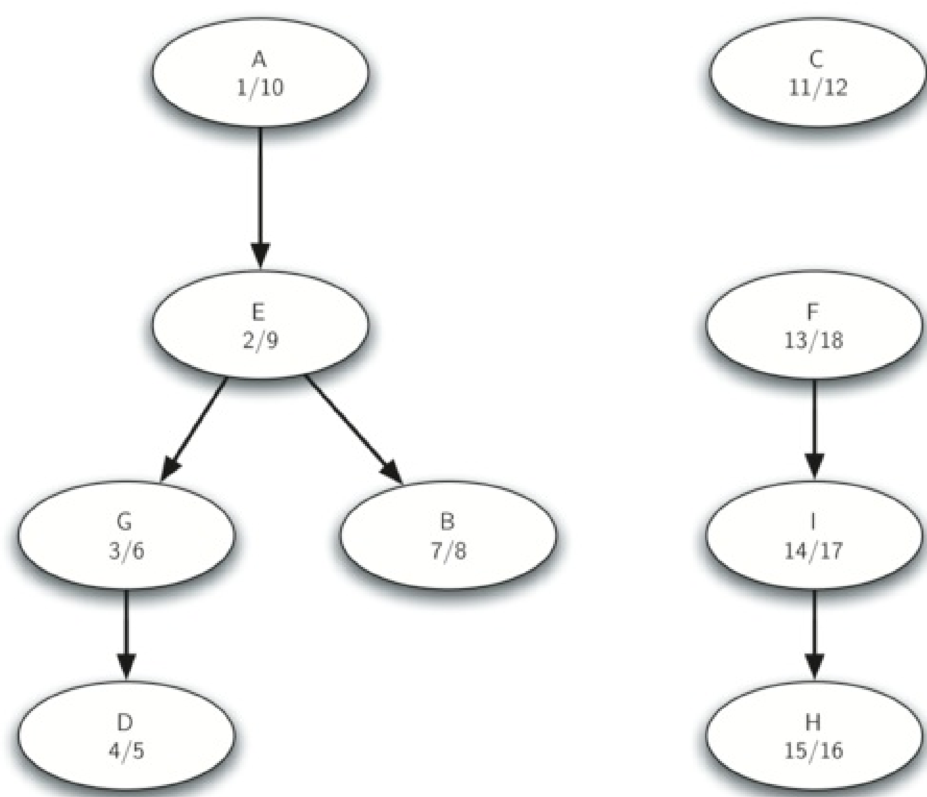



*Figure 37*

# 7.19.最短路径问题

当你在网上冲浪，发送电子邮件，或从校园的另一个地方登录实验室计算机时，大量的工作正在幕后进行，以获取你计算机上的信息传输到另一台计算机。 深入研究信息如何通过互联网从一台计算机流向另一台计算机是计算机网络中的一个主要课题。 然而，我们将讨论互联网如何工作足以理解另一个非常重要的图算法。

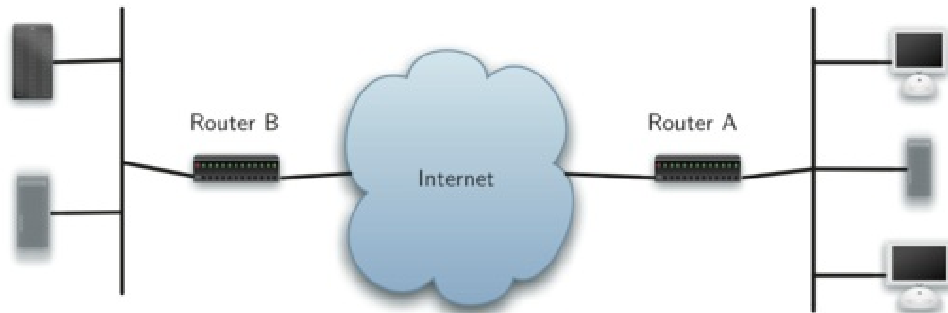



*Figure 1*

Figure 1 展示了 Internet 上的通信如何工作的高层概述。当使用浏览器从服务器请求网页时，请求必须通过局域网传输，并通过路由器传输到 Internet上。 该请求通过因特网传播，并最终到达服务器所在的局域网路由器。 请求的网页然后通过相同的路由器回到您的浏览器。 在 Figure 1中标记为“因特网”的云是附加的路由器。所有这些路由器一起工作，让信息从一个地方到另一个地方。 可以看到有许多路由器，如果你的计算机支持 `traceroute` 命令。下面的文本显示 `traceroute` 命令的输出，说明在 `Luther College` 的Web服务器和明尼苏达大学的邮件服务器之间有13个路由器

```
1  192.203.196.1
2  hilda.luther.edu (216.159.75.1)
3  ICN-Luther-Ether.icn.state.ia.us (207.165.237.137)
4  ICN-ISP-1.icn.state.ia.us (209.56.255.1)
5  p3-0.hsa1.chi1.bbnplanet.net (4.24.202.13)
6  ae-1-54.bbr2.Chicago1.Level3.net (4.68.101.97)
7  so-3-0-0.mpls2.Minneapolis1.Level3.net (64.159.4.214)
8  ge-3-0.hsa2.Minneapolis1.Level3.net (4.68.112.18)
9  p1-0.minnesota.bbnplanet.net (4.24.226.74)
10  TelecomB-BR-01-V4002.ggnet.umn.edu (192.42.152.37)
11  TelecomB-BN-01-Vlan-3000.ggnet.umn.edu (128.101.58.1)
12  TelecomB-CN-01-Vlan-710.ggnet.umn.edu (128.101.80.158)
13  baldrick.cs.umn.edu (128.101.80.129)(N!)  88.631 ms (N!)


Routers from One Host to the Next over the Internet
```

互联网上的每个路由器都连接到一个或多个路由器。因此，如果在一天的不同时间运行 traceroute 命令，你很可能会看到你的信息在不同的时间流经不同的路由器。这是因为存在与一对路由器之间的每个连接相关联的成本，这取决于业务量，一天中的时间以及许多其他因素。到这个时候，你不会惊讶，我们可以将路由器的网络表示为带有加权边的图形。

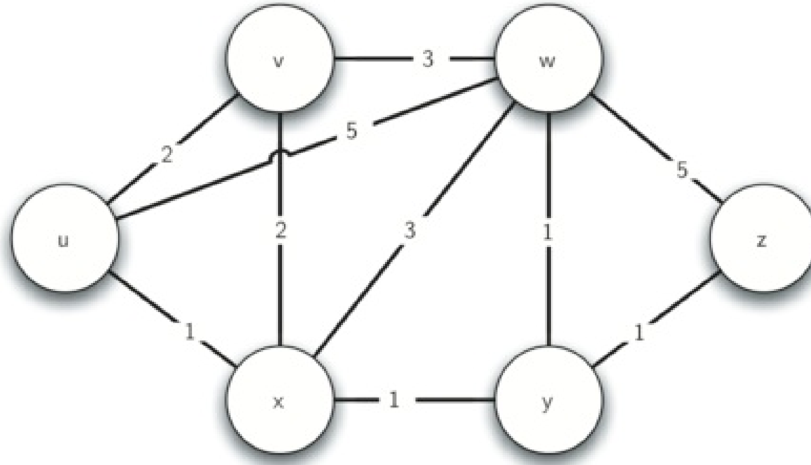



*Figure 2*

Figure 2 展示了表示互联网中的路由器的互连的加权图的一个小例子。我们要解决的问题是找到具有最小总权重的路径，沿着该路径路由传送任何给定的消息。这个问题听起来很熟悉，因为它类似于我们使用广度优先搜索解决的问题，我们这里关心的是路径的总权重，而不是路径中的跳数。应当注意，如果所有权重相等，则问题是相同的。

# 7.20.Dijkstra算法

我们将用于确定最短路径的算法称为“Dijkstra算法”。Dijkstra算法是一种迭代算法，它为我们提供从一个特定起始节点到图中所有其他节点的最短路径。这也类似于广度优先搜索的结果。

为了跟踪从开始节点到每个目的地的总成本，我们将使用顶点类中的 dist 实例变量。 dist实例变量将包含从开始到所讨论的顶点的最小权重路径的当前总权重。该算法对图中的每个顶点重复一次;然而，我们在顶点上迭代的顺序由优先级队列控制。用于确定优先级队列中对象顺序的值为dist。当首次创建顶点时，dist被设置为非常大的数。理论上，你将 dist 设置为无穷大，但在实践中，我们只是将它设置为一个数字，大于任何真正的距离，我们将在问题中试图解决。

Dijkstra算法的代码如 Listing 1 所示。当算法完成时，距离设置正确，如图中每个顶点的前导链接一样

```
from pythonds.graphs import PriorityQueue, Graph, Vertex
def dijkstra(aGraph,start):
    pq = PriorityQueue()
    start.setDistance(0)
    pq.buildHeap([(v.getDistance(),v) for v in aGraph])
    while not pq.isEmpty():
        currentVert = pq.delMin()
        for nextVert in currentVert.getConnections():
            newDist = currentVert.getDistance() \
                    + currentVert.getWeight(nextVert)
            if newDist < nextVert.getDistance():
                nextVert.setDistance( newDist )
                nextVert.setPred(currentVert)
                pq.decreaseKey(nextVert,newDist)
```

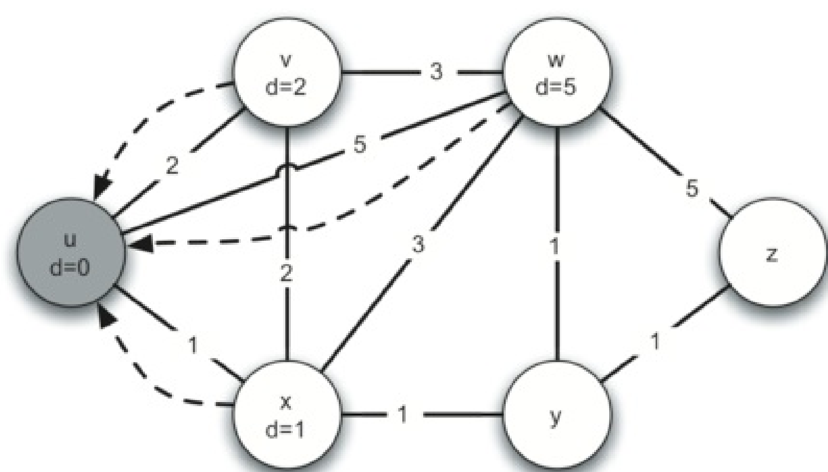
*Listing 1*

Dijkstra的算法使用优先级队列。你可能还记得，优先级队列是基于我们在树章节中实现的堆。这个简单的实现和我们用于Dijkstra算法的实现之间有几个区别。首先，PriorityQueue 类存储键值对的元组。这对于Dijkstra的算法很重要，因为优先级队列中的键必须匹配图中顶点的键。其次，值用于确定优先级，并且用于确定键在优先级队列中的位置。在这个实现中，我们使用到顶点的距离作为优先级，因为我们看到当探索下一个顶点时，我们总是要探索具有最小距离的顶点。第二个区别是增加 `decreaseKey` 方法。正如你看到的，当一个已经在队列中的顶点的距离减小时，使用这个方法，将该顶点移动到队列的前面。

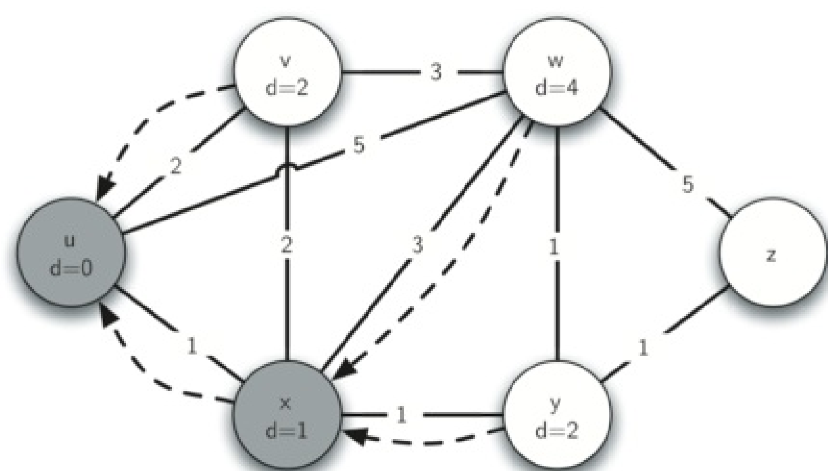
让我们使用下面的序列图像作为指导，一次应用 Dijkstra 算法的一个顶点。我们从顶点 u 开始。与 u 相邻的三个顶点是 v，w 和 x。由于到 v，w 和 x 的初始距离都被初始化为 `sys.maxint`，通过起始节点获得它们的新成本都是它们的直接成本。因此，我们将这三个节点中的每一个成本更新。我们还将每个节点的前导设置为 u，并将每个节点添加到优先级队列。我们使用距离作为优先级队列的键。算法的状态如 Figure 3所示。

在 while 循环的下一次迭代中，我们检查与 x 相邻的顶点。顶点 x 是下一个，因为它具有最低的总成本，因此冒泡到优先级队列的开始。在 x，我们看看它的邻居 u，v，w 和 y。对于每个相邻顶点，我们检查通过 x 到该顶点的距离是否小于先前已知的距离。显然，这是 y 的情况，因为它的距离是 sys.maxint。这不是 u 或 v 的情况，因为它们的距离分别为 0 和 2。然而，我们现在知道，如果我们经过 x 而不是从 u 直接到 w，到 w 的距离更小。既然是这样，我们用新的距离更新 w，并将 w 的前导从 u 更改为 x。有关所有顶点的状态，请参见 Figure 4。

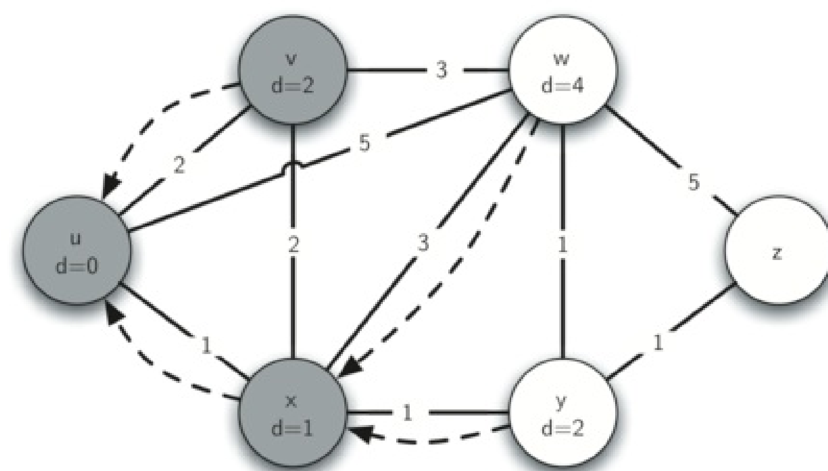
下一步是查看邻近 v 的顶点（参见 Figure 5）。此步骤不会对图形进行任何更改，因此我们继续前进到节点 y。在节点 y（见Figure 6），我们发现到 w 和 z 都更小，因此我们相应地调整距离和前导链接。最后，我们检查节点 w 和 z（参见 Figure 6 和 Figure 8）。但是，没有发现额外的更改，因此优先级队列为空，Dijkstra的算法退出。



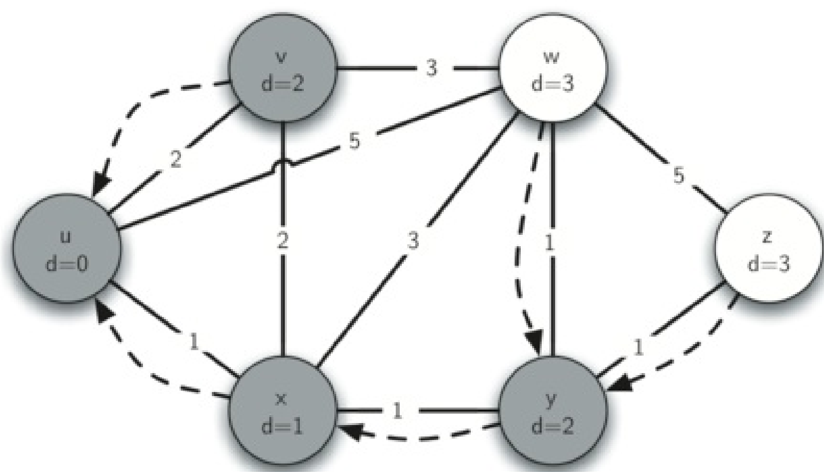

PQ = x,v,w



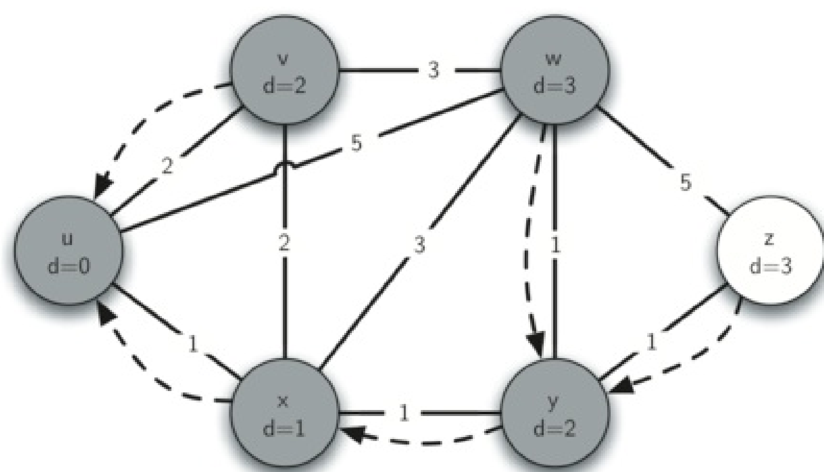

PQ = vy,w



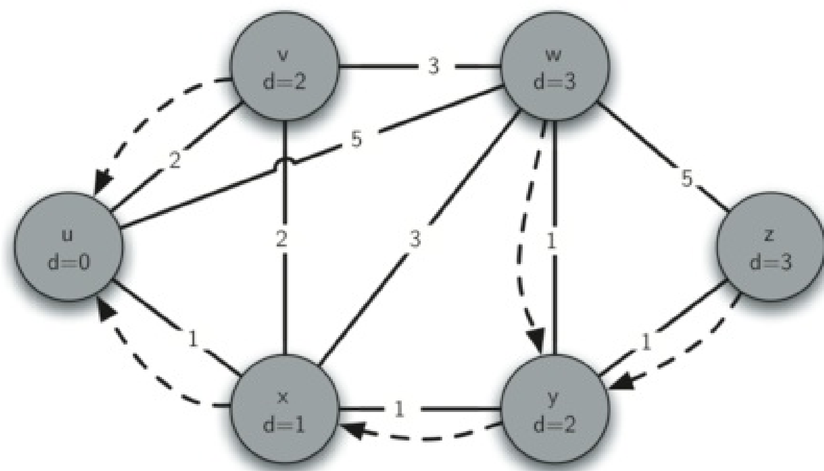

PQ = yw




PQ = wz




PQ = z




PQ = None

重要的是要注意，Dijkstra的算法只有当权重都是正数时才起作用。 如果你在图的边引入一个负权重，算法永远不会退出。

我们将注意到，通过因特网路由发送消息，可以使用其他算法来找到最短路径。 在互联网上使用 Dijkstra 算法的一个问题是，为了使算法运行，你必须有一个完整的图表示。 这意味着每个路由器都有一个完整的互联网中所有路由器地图。 实际上不是这种情况，算法的其他变种允许每个路由器在它们发送时发现图。 你可能想要了解的一种这样的算法称为“距离矢量”路由算法。

# 7.21.Dijkstra算法分析

最后，让我们看看 Dijkstra 算法的运行时间。我们首先注意到，构建优先级队列需要 O(v) 时间，因为我们最初将图中的每个顶点添加到优先级队列。 一旦构造了队列，则对于每个顶点执行一次 while 循环，因为顶点都在开始处添加，并且在那之后才被移除。 在该循环中每次调用 delMin，需要 O(log^V )时间。 将该部分循环和对 delMin 的调用取为 O(Vlog^V )。 for 循环对于图中的每个边执行一次，并且在 for 循环中，对 decreaseKey 的调用需要时间 O(Elog^V) 。 因此，组合运行时间为 O((V + E)log^V )。

# 7.22.Prim生成树算法

对于我们最后的图算法，让我们考虑一个在线游戏设计师和网络收音机提供商面临的问题。问题是他们想有效地将一条信息传递给任何人和每个可能在听的人。 这在游戏中是重要的，使得所有玩家知道每个其他玩家的最新位置。 对于网络收音机是重要的，以便所有该调频的收听者获得他们需要的所有数据来刷新他们正在收听的歌曲。 Figure 9 说明了广播问题。

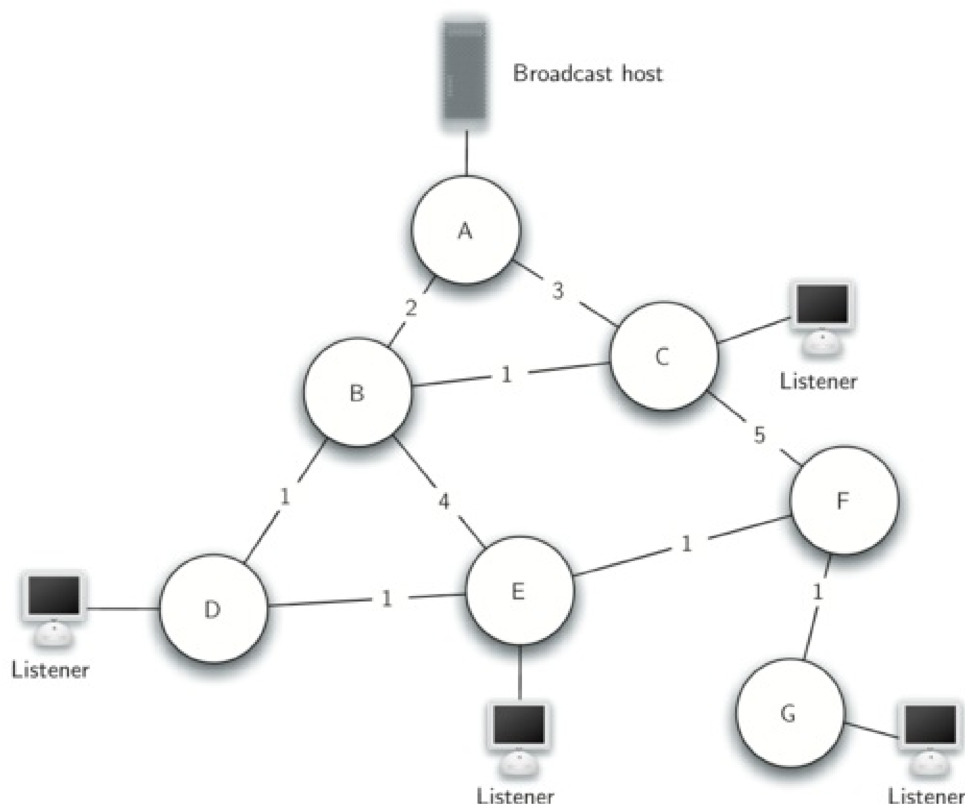



*Figure 9*

这个问题有一些强力的解决方案，所以先看看他们如何更好地理解广播问题。这也将帮助你理解我们最后提出的解决方案。首先，广播主机有一些收听者都需要接收的信息。最简单的解决方案是广播主机保存所有收听者的列表并向每个收听者发送单独的消息。在 Figure 9 中，我们展示了有广播公司和一些收听者的小型网络。使用第一种方法，将发送每个消息的四个副本。假设使用最小成本路径，让我们看看每个路由器处理同一消息的次数。

来自广播公司的所有消息都通过路由器 A，所以 A 看到每个消息的所有四个副本。路由器 C 只接收到其收听者每个消息的一个副本。然而，路由器 B 和 D 将收到每个消息的三个副本，因为路由器 B 和 D 在收听者 1,2和3 的最短路径上。当广播主机必须每秒发送数百条消息用于无线电广播，这是很多额外的流量。

暴力解决方案是广播主机发送广播消息的单个副本，并让路由器整理出来。在这种情况下，最简单的解决方案是称为 `不受控泛洪` 的策略。洪水策略工作如下。每个消息开始于将存活时间（ttl）值设置为大于或等于广播主机与其最远听者之间的边数量的某个数。每个路由器获得消息的副本，并将消息传递到其所有相邻路由器。当消息传递到 ttl 减少。每个路由器继续向其所有邻居发送消息的副本，直到 ttl 值达到 0。不受控制的洪泛比我们的第一个策略产生更多的不必要的消息。

这个问题的解决方案在于建立最小权重 `生成树` 。正式地，我们为图 G =（V，E）定义最小生成树 T 如下。 T 是连接 V 中所有顶点的 E 的非循环子集。 T中的边的权重的和被最小化。

Figure 10 展示了广播图的简化版本并突出了生成图的最小生成树的边。现在为了解决我们的广播问题，广播主机简单地将广播消息的单个副本发送到网络中。每个路由器将消息转发到作为生成树的一部分邻居，排除刚刚向其发送消息的邻居。在这个例子中 A 将消息转发到 B，B 将消息转发到 D 和 C。D 将消息转发到 E，E将它转发到 F，F 转发到 G。没有路由器看到任何消息的多个副本，所有感兴趣的收听者都会看到消息的副本。

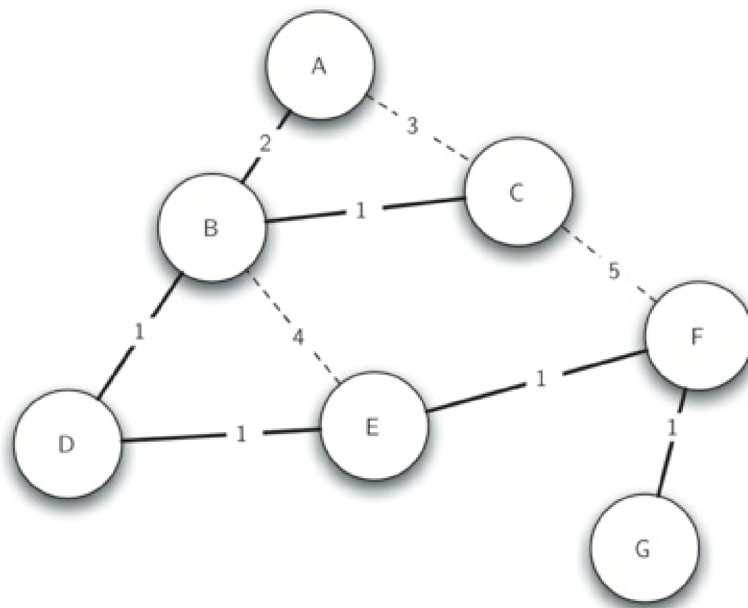



*Figure 10*

我们将用来解决这个问题的算法称为 Prim 算法。 Prim 算法属于称为 “贪婪算法” 一系列算法，，因为在每个步骤，我们将选择最小权重的下一步。 在这种情况下，最小权重的下一步是以最小的权重跟随边。 我们的最后一步是开发 Prim 算法。

构建生成树的基本思想如下：

```
While T is not yet a spanning tree
   Find an edge that is safe to add to the tree
   Add the new edge to T
```

诀窍是指导我们“找到一个安全的边”。我们定义一个安全边作为将生成树中的顶点连接到不在生成树中的顶点的任何边。这确保树将始终保持为树并且没有循环。

用于实现 Prim 算法的 Python 代码如 Listing2 所示。Prim 算法类似于Dijkstra 算法，它们都使用优先级队列来选择要添加到图中的下一个顶点。

```
from pythonds.graphs import PriorityQueue, Graph, Vertex

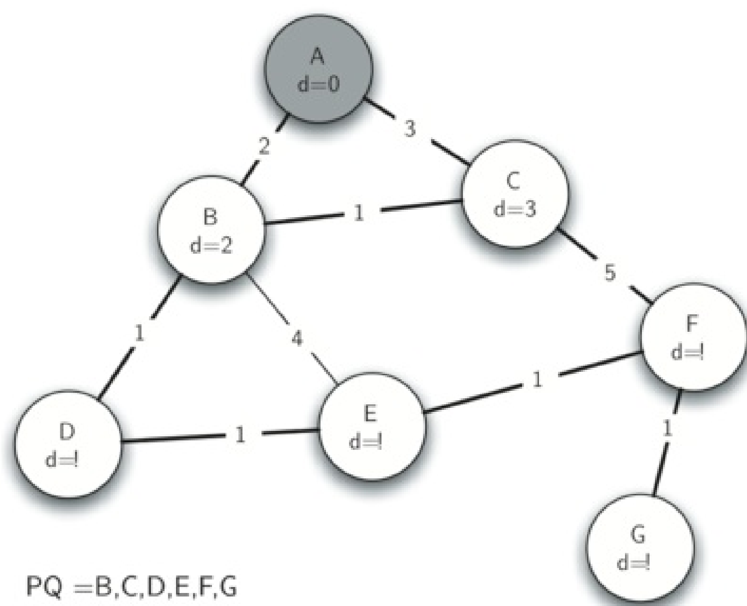
def prim(G,start):
    pq = PriorityQueue()
    for v in G:
        v.setDistance(sys.maxsize)
        v.setPred(None)
    start.setDistance(0)
    pq.buildHeap([(v.getDistance(),v) for v in G])
    while not pq.isEmpty():
        currentVert = pq.delMin()
        for nextVert in currentVert.getConnections():
          newCost = currentVert.getWeight(nextVert)
          if nextVert in pq and newCost<nextVert.getDistance():
              nextVert.setPred(currentVert)
              nextVert.setDistance(newCost)
              pq.decreaseKey(nextVert,newCost)
```

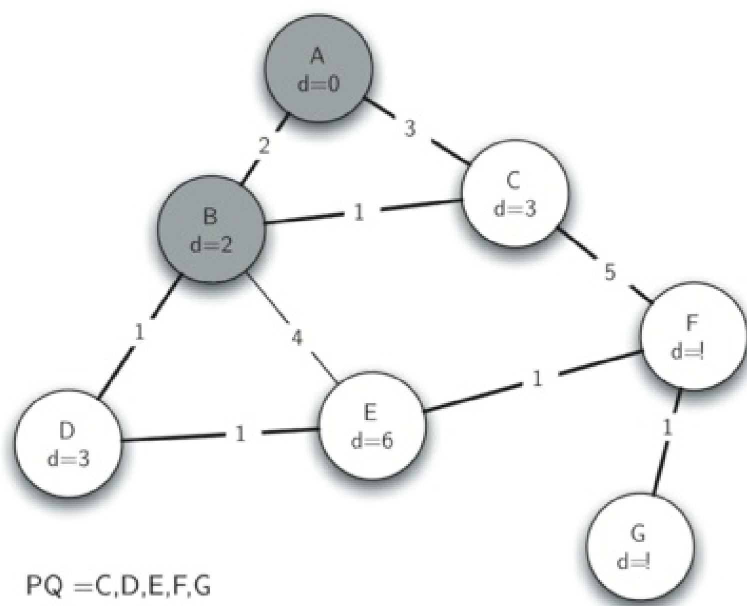
*Listing 2*

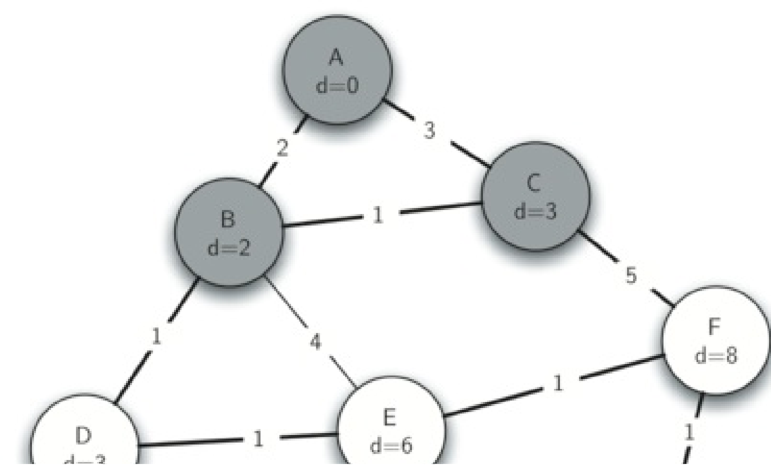
下面的图（Figure 11 到 Figure 17）展示了在我们的样本树上运行的算法。我们从起始顶点开始。到所有其他顶点的距离被初始化为无穷大。看看 A 的邻居，我们可以更新另外两个顶点 B 和 C 的距离，因为通过 A 到 B 和 C 的距离小于无限。这将 B 和 C 移动到优先级队列的前面。通过将 B 和 C 的前导链接设置为指向 A 来更新前导链接。重要的是要注意，我们还没有正式向生成树添加 B 或 C。在将节点从优先级队列中删除之前，不会将其视为生成树的一部分。

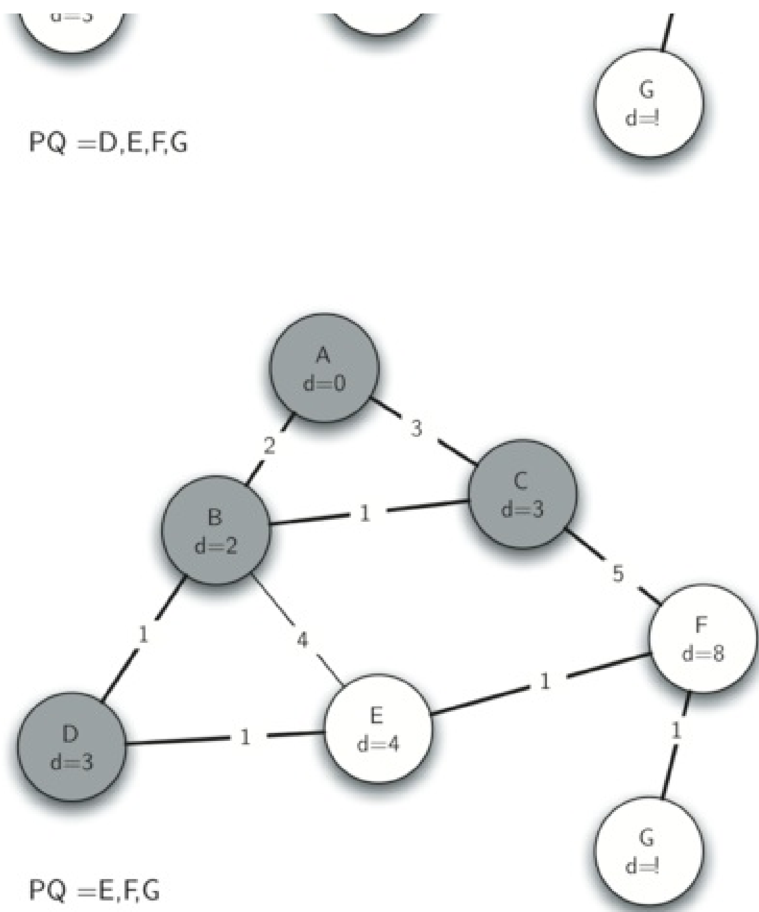
因为 B 有最小的距离，我们看看 B。检查 B 的邻居，我们看到 D 和 E 可以更新。D 和 E 都获得新的距离值，并更新它们的前导链接。移动到优先级队列中的下一个节点，我们找到 C。只有仍在优先级队列中的节点是 F，因此我们可以更新到 F 的距离，并调整优先级队列中的 F 的位置。

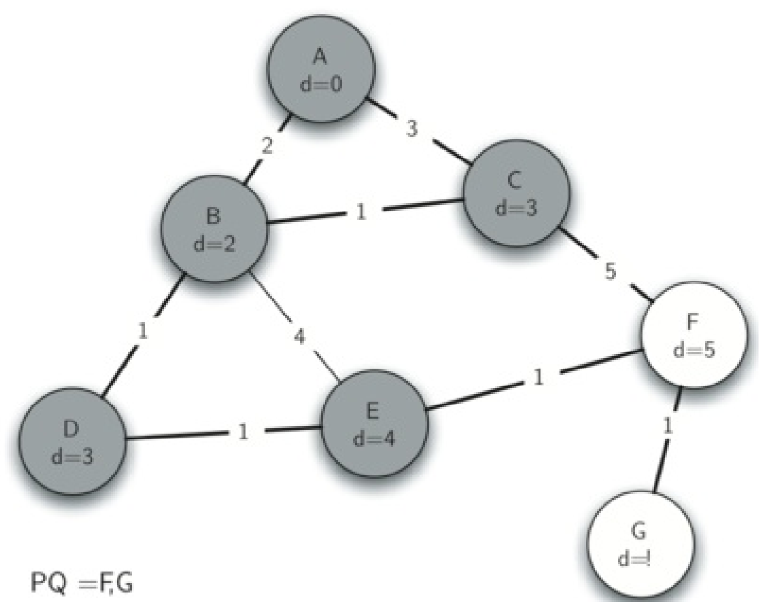
现在我们检查与节点 D 相邻的顶点。我们发现可以更新 E 并且将从距离 6 减小到 4。当我们这样做时，我们将 E 上的前趋链接改变为指向 D，从而准备移植到生成树中不同的位置。算法的其余部分按照预期进行，将每个新节点添加到树中。

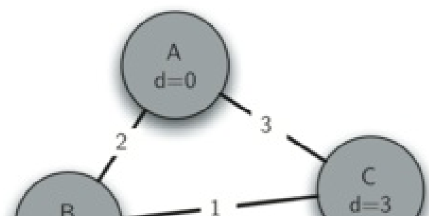

A
d=0
3
2
C
d=3
B
d=2
1
5
F
d=!
1
4
1
E
d=!
D
d=!
1
1
G
d=!
PQ =B,C,D,E,F,G

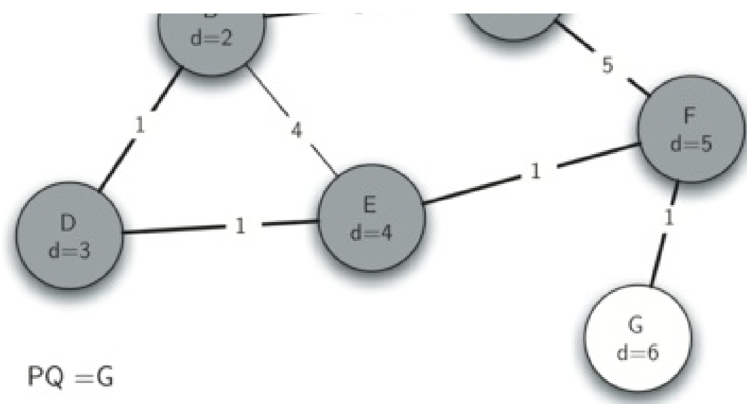

A
d=0
3
2
C
d=3
B
d=2
1
5
F
d=!
1
4
1
E
d=6
D
d=3
1
1
G
d=!
PQ =C,D,E,F,G

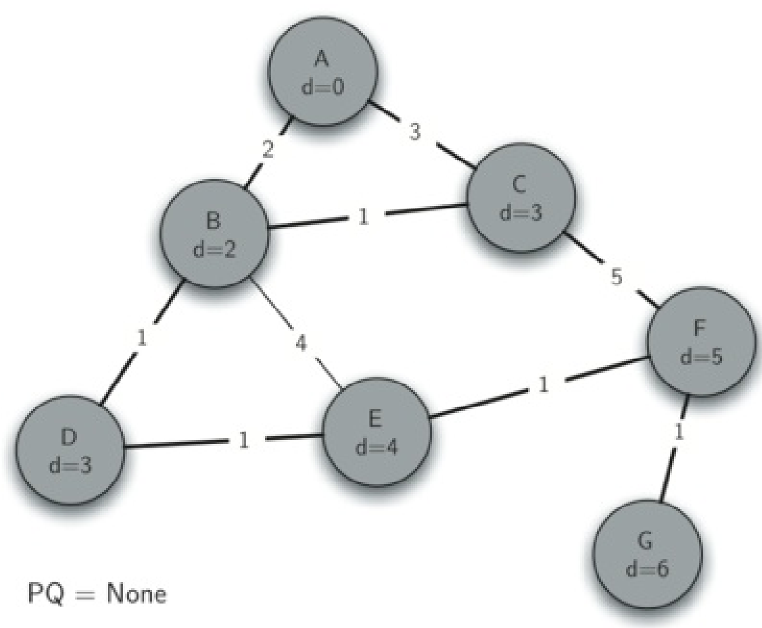

A
d=0
3
2
C
d=3
B
d=2
1
5
F
d=8
1
4
1
E
d=6
D
1
1


G
d=!
PQ =D,E,F,G
A
d=0
B
d=2
C
d=3
D
d=3
E
d=4
F
d=8
G
d=!
PQ =E,F,G


A
d=0
B
d=2
C
d=3
D
d=3
E
d=4
F
d=5
G
d=!
PQ =F,G


A
d=0
C
d=3


d=2
5
1
4
F
d=5
D
d=3
1
E
d=4
1
1
G
d=6
PQ =G


A
d=0
2
3
C
d=3
B
d=2
1
5
1
4
F
d=5
1
D
d=3
1
E
d=4
1
G
d=6
PQ = None

# 7.23.总结

在本章中，我们讨论了图抽象数据类型，以及图的一些实现。 图使我们能够解决许多问题，只要我们可以将原始问题转换为可以由图表示的东西。 特别是，我们已经看到，图有助于解决以下领域的问题。

- 广度优先搜索找到未加权的最短路径。
- Dijkstra的加权最短路径算法。
- 深度优先搜索图探索。
- 强连通分量，用于简化图。
- 排序任务的拓扑排序。
- 广播消息的最小权重生成树。